A Dynamic Prediction and Monitoring Framework for Distributed Applications

by

James David Turner

A thesis submitted to the University of Warwick

in partial fulfilment of the requirements

for admission to the degree of

Doctor of Philosophy

Department of Computer Science

University of Warwick

May 2003

Abstract

This research builds on an application performance prediction and characterisation environment (known as PACE), whose aim is to characterise the performance-critical elements of both an application and its target execution environment and deduce from this model a predicted behaviour of the application prior to its execution.

Underlying the research presented in this thesis are a number of themes: the tasks involved in the performance characterisation of applications and how this might be semi-automated; the level of abstraction at which these characterisations are performed in order to maintain a sufficient predictive accuracy; the automated refinement of these characterisations from runtime performance data; the extension of both the target programming languages and the class of application at which these techniques are aimed.

In this thesis a number of novel extensions to PACE are described. These include: a new transaction-based performance characterisation language that provides a flexible framework for describing broader classes of application; a performance monitoring framework (based on an extension to the OpenGroup's Application Response Measurement (ARM) standard) for the runtime monitoring of an application's data-dependent components and the automated refinement of performance models; an adaptation of this performance characterisation for the prediction of Java applications. These contributions are demonstrated through their application to a number of scientific kernels. This thesis also documents how these predictive results can be used in a real-time distributed runtime management environment, and also how these techniques can be applied to non-scientific codes, in particular to an IBM request-driven distributed web services demonstrator.

Contents

List of Figures

List of Listings

List of Graphs

List of Tables

Acknowledgments

I have been thinking for a while now whether there has been another part of my life that comes close to the emotional roller-coaster that I have experienced during the course of writing this thesis. Although winning the University of Warwick Table Football cup final does come close, I am still searching.

I do not believe that I could have finished this degree without the overwhelming support of the following people. Graham Nudd: chairman of the research group and department, and a continuing inspiration to me over the last three years. Daniel Spooner: good friend, fellow researcher and partner in crime. Stephen Jarvis: supervisor, advisor and friend, who has gone beyond the call of duty on many occasion. My parents David and Karen and brother Stuart who have provided the perfect start in life. Sam: girlfriend and best friend; thanks for waiting.

I would also like to thank the following people who are close to me: members past and present of the High Performance Systems Group including David Bacigalupo, Junwei Cao, Trevor Chambers, Darren Kerbyson, Helene Lim Choi Keung, Justin Dyson, Ligang He, Lei Zhao, Nathan Griffiths, John Harper and Stuart Perry; members of the Department of Computer Science including Roger Packwood, Franc Buxton, Gerry Biggs and Rod Moore; members of the Residential Tutor system including John Cunnington, Diarmuid McAuliffe and Ken Sloan; and of course JMBF.

Declarations

This thesis is presented in accordance with the University of Warwick regulations for the degree of Doctor of Philosophy. It has been written by myself and has not been submitted in any previous application for any degree at any other institution. The work documented in this thesis has been undertaken by myself except where otherwise stated.

The various aspects of research that are documented in this thesis have also been published in the following publications: [Spooner01], [Turner01], [Cao02b], [Cao02a], [Spooner02a], [Turner02a], [Turner02b], [Turner02c], {Bacigalupo03a] and [Bacigalupo03b].

Abbreviations

A4	Agile Architecture and Autonomous Agents
ACT	Automated Characterisation Tool
API	Application Program Interface
ARM	Application Response Measurement
ASCI	Advanced Simulation and Computing Program
B2B	Business to Business
BCEL	Byte Code Engineering Library
CHIP^{3}S	Characterisation Instrumentation for the Performance Prediction of Parallel Systems
CORBA	Common Object Request Broker Architecture
CPU	Central Processing Unit
DaSSF	Dartmouth Scalable Simulation Framework
DERA	Defense Electronic Research Agency
DML	Domain Modelling Language
FFT	Fast Fourier Transform
GMA	Grid Monitoring Architecture
HLFD	Hierarchical Layered Framework Diagram
HTTP	Hypertext Transfer Protocol
IBM	International Business Machines
IDE	Integrated Development Environment
IDEA	International Data Encryption Algorithm

JAMM	Java Agents for Monitoring and Management
jPACE	java Performance Analysis and Characterisation Environment
jPCL	jPACE Performance Characterisation Language
JVM	Java Virtual Machine
MPI	Message Passing Interface
NAS	NASA Advanced Supercomputing
NASA	National Aeronautics and Space Administration
OGSA	Open Grid Services Architecture
PACE	Performance Analysis and Characterisation Environment
PASE	Performance Analysis and Simulation Environment
PDA	Personal Digital Assistant
POEMS	Performance Oriented End-to-end Modelling System
QoS	Quality of Service
RMI	Remote Method Invocation
RPPT	Rice Parallel Processing Testbed
SCALA	Scalability Analyser
SLA	Service Level Agreement
SMC	Spatial Motion Compression
SPEC	Standard Performance Evaluation Corporation
UDDI	Universal Description, Discovery and Intergration
UUID	Universally Unique Identifiers
WAN	Wide Area Network

Chapter 1

Introduction

Ever since computing resources have been used to address scientific and engineering problems more efficiently, application developers have spent a great deal of time developing the infrastructures necessary to achieve the greatest possible performance. The methodology and philosophy behind such high performance computing was, and continues to be, the creation of massively parallel supercomputers, which has resulted in expensive, high maintenance systems with hundreds of processors and terabytes of local storage. Applications that are deployed on such resources typically consist of a number of highly optimised mathematical kernels, executed repeatedly and in parallel on very large input data sets, that can, in some cases, require several months to conclude.

Choosing a suitable hardware platform for a parallel application to achieve the greatest possible performance is not a simple issue. Different kernels execute efficiently on different hardware with different configurations (memory, number of processors and so on). It is not always the case that allocating as much hardware as possible to a problem is a suitable response, or that it will result in the greatest performance; executing a kernel on an increasing number of processors may decrease its performance as communication outweighs the benefits of increasing computational ability. Performance evaluation and optimisation is therefore of major interest within

the high-performance community for providing optimum solutions for parallel applications.

1.1 Performance Evaluation and Prediction

Over the last ten years, a number of tools have been developed at the University of Warwick to provide parallel application developers with the ability to evaluate and predict the performance of their programs. The main contribution to this research, the Performance Analysis and Characterisation Environment (PACE) [Cao00, Kerbyson96, Kerbyson98a, Nudd00], enable developers to characterise the performance of both the parallel applications and the hardware that those applications will be executed upon. Characterisations are then compiled into a performance model that can be rapidly evaluated to provide developers with detailed predictions regarding the performance of an application's execution on a variety of hardware configurations.

Application characterisations in PACE consist of a layered framework [Nudd93, Papaefstathiou94, Papaefstathiou97] that includes several performance objects written in a characterisation language called CHIP^{3}S [Papaefstathiou95a, Papaefstathiou95b]. Each performance object describes a specific performance-critical element of the system, whether that be a sequential area of computation, how these sequential areas are parallelised among the available hardware, the communication between these computations, or the hardware itself. Each object is parameterised so that factors such as input data size and the number of processors can be adjusted so that data- and processor-scalability analysis of the application can be explored. Performance evaluations have been verified by the Defense Electronic Research Agency (DERA) to have a prediction error of less than 10% [Nudd00].

The ability to predict an application's performance prior to execution not only provides a developer with valuable insight into performance-critical areas of his/her application, but can also be embedded within middleware to facilitate the mapping of

applications to the available hardware and resources. Two examples include:

1. Application steering [Alkindi00, Kerbyson98b, Turner02c], where an application has the ability to predict its own performance. With this ability, and given an environmental constraint of some kind, the application can predict its future performance prior to execution and, by using this information, choose a correct set of application parameters and resource such that the constraint is met[1].

2. Application scheduling, where a number of submitted applications are to be efficiently allocated to a known set of available resources. The overall throughput of applications within a distributed environment can be dramatically increased if their performance is predicted and taken into account by the scheduling algorithm [Jarvis03b].

1.2 Grid Computing

More recently, the methodology behind high-performance computing has changed. With the vast increase in network bandwidth and desktop computer performance, along with the reduction in price of these resources, both academic and corporate research has focused on distributed, heterogeneous environments as the next-generation platform-of-choice for both e-science (highly-computational scientific codes that use very large data collections, terascale computing resources and high performance visualisation) and e-business (user-driven, high-throughput, transaction-based) applications. This is demonstrated most notably in the emergence of Grid computing [Foster98, Foster01, Leinberger99]; geographically-dispersed, resource-sharing networks and disparate, dynamic, heterogeneous resources, whose management, rather than being centralised, is maintained through multiple administrative domains that span multiple institutions,

[1] application steering differs from application reflection [Huang01, Kon00] in that a reflective application, instead of committing to a specific behaviour initially, can continuously adapt to the environment dynamically during execution

countries, and continents. Such an environment would be able to provide 'computation-on-demand', with the source of the computation and the infrastructure present to provide it being transparent to the user.

Implementing a Grid architecture is a complex task. In order to support the Grid philosophy, it is necessary to provide sophisticated middleware services that can operate efficiently within these environments. The Globus toolkit [Foster97], version three of which implements the Open Grid Services Architecture (OGSA) [Foster02a], is becoming a standard for Grid service and platform development. Globus consists of a number of 'Grid-enabled' applications and APIs to allow developers to design Grid applications, as well as dynamic and scalable services for information storage, application submission, security and authentication. The outcome is a transparent middleware layer that can provide a sustainable, reliable and predictable computing service, irrespective of variations in the available resources and user demands.

More recently, Warwick has concentrated on the research and development of a number of frameworks that complement Globus and other standard Grid middleware to efficiently schedule applications among available resources. TITAN [Spooner02a, Spooner02b], a resource scheduler for distributed architectures, is a significant component of this research. TITAN uses iterative heuristic algorithms to optimise a time-line of scheduled applications while aiming to improve a number of scheduling metrics (such as makespan and idle time) and continuing to meet quality of service (QoS) metrics (including deadline time, the percentage of tasks meeting this deadline, resource balance/usage and user/application priority [Jarvis03b]). Intra-domain resource management is implemented as an additional Grid middleware service that efficiently directs the Condor scheduler [Litzkow88] (a popular scheduler used within Grid environments), while a peer-to-peer agent management system interfaces with the Globus information services [Foster02b, Keung02a, Keung02b] to support resource advertisement and discovery.

TITAN is differentiated from other Grid scheduling frameworks by its application of the following:

- Resource Discovery and Advertisement using A4. Agile Architecture and Autonomous Agents (A4) [Cao01c, Cao01b, Cao01a] is an agent-based methodology for building large-scale distributed software systems with highly-dynamic behaviour. Each heterogeneous resource is described by a homogeneous agent that contains a repository of performance information regarding the local and neighbouring resources. New resources added to the environment are discovered dynamically by A4 and advertised to neighbouring agents, providing an infrastructure where a suitable resource can be found for the execution of a scheduled application.

  A4 provides TITAN with an awareness of other local resources within its agent hierarchy, supplying an infrastructure for resource management in the scheduling of applications across administrative domains. Applications whose QoS requirements would not be met if they were executed on the cluster where submission took place can be moved to a more suitable resource within the agent environment. It has been shown in [Jarvis03a, Spooner03] that this inter-domain resource management can provide an 80% improvement over a more common 'first-come first-served' execution.

- Predicted Application Performance using PACE. Predictive performance data is used by TITAN to optimise both the scheduling time-line at the cluster level and the management of resources at the inter-domain level. An application is submitted to TITAN with its PACE performance model. This model is then evaluated by TITAN prior to execution so that the optimum set of resources for that application can be chosen in order to meet the user's requirements. Iteratively predicting and refining the scheduling time-line before the execution of all applications allows TITAN to choose an optimum schedule for submitted applications.

In [Foster98] it is stated that Grid computing will provide the computing infrastructure for the following classes of application:

- *Distributed Supercomputing*, which involves very large problems that require large amounts of computing power;
- *High-Throughput Computing*, which harnesses many otherwise idle resources to increase aggregate throughput;
- *On-Demand Computing*, where remote resources integrate with local computation, often for a specific amount of time;
- *Data-Intensive Computing*, which involves the synthesis of new information from many or large data sources;
- *Collaborative Computing*, which supports communication or collaborative work between multiple participants.

The research in this thesis is focused on the performance prediction of the 'High-Throughput Computing' class of applications, in particular, MPI-based, scientific applications. This work has been designed however in order to be easily extended to other classes of application as described above. These extensions are the subject of future work.

1.3 Contributions of this Thesis

Providing the ability to model and predict the performance of applications can be invaluable both for developers to optimise their applications and for middleware services (such as TITAN) to achieve efficient resource allocation within distributed environments. However, while PACE can provide accurate performance evaluations of scientific applications executed on static hardware platforms, there are aspects of its implementation that make PACE less suitable for the dynamic nature of Grid architectures:

- PACE requires an application to be fully characterised before evaluation. It is therefore necessary for the complete performance model to be developed and compiled before any predictions can be made, which requires time and a fundamental knowledge of creating PACE performance models. Should a quick performance evaluation be necessary, this requirement may be unacceptable. There is currently no way within PACE to evaluate incomplete characterisations in order to obtain a (perhaps) less accurate predictive result.

- PACE achieves performance predictions by evaluating static performance models upon static hardware characterisations. If a performance model was found to be inaccurate, due to changing data sets or resource load fluctuations for example, the application characterisation and/or hardware model would have to be modified and the model recompiled before more accurate predictions could take place. Currently there is not an efficient way within PACE to automate the refinement of performance models 'on-the-fly', in order to accommodate this dynamic behaviour and provide a basis for predicting data-dependent code.

- PACE characterises the performance of sequential elements of computation from the original source code. Atomic measurements of machine-code instructions are extrapolated from the source code and evaluated against hardware benchmarks to provide predictive results. This may result in inaccuracies between the characterised computation at the source code level and any optimisations that may have been made by the compiler, as well as the requirement of having access to the application's source code in the first place. There is currently no way within PACE to characterise computation at the object code level.

- PACE characterises the application's eventual hardware as a single hardware object which describes the resource's computation and inter-resource communication. However, the execution environment of some modern applications can be far more complex and involve many more performance-critical elements that

cannot be characterised within a single hardware object. For example, the performance of virtual machine-based applications is significantly affected by the virtual machine implementation, web-based applications are affected by web-server loads, and so on. There is currently no way within PACE to characterise and evaluate these performance-critical elements of the application's execution environment.

In response to these issues, this thesis describes a number of proposed extensions to PACE and the details regarding their implementation:

- Instead of an application characterised as a set of sequential elements, each with an associated description of how each element executes among a set of homogeneous resources, a flexible level of performance characterisation is introduced. Applications are characterised as a control-flow of transactions, with the relationship between these transactions described within a transaction map. Each transaction characterises an item of work which can be either a sequential computation, an inter-resource communication, or a combination of the two. Furthermore, a transaction's characterisation can range from a simple description of the item of work (a class's method call or even an entire application) to a full control-flow analysis of the application's object code (similar to PACE). This flexible method of performance characterisation can result in performance evaluations from either resource benchmark measurements or historical performance data, recorded by the monitoring framework. This flexibility provides the notion of a trade-off between the model's level of characterisation detail (and, inherently, model creation time) and the eventual predictive accuracy, as well as providing the basis for the ability to characterise a greater selection of application.

- In order to extend PACE into a more portable framework for use within a heterogeneous environment, performance characterisations are described within a newly developed, XML-based language, and the evaluation engine has been

re-implemented for the Java platform. A performance model is comprised of a collection of XML-based performance objects, that can be evaluated on any platform that includes a Java Virtual Machine (JVM). Furthermore, XML-based performance models, being text-based, can be highly compressed for communication within a Grid environment, as well as easily instrumented or modified without any re-compilation or linking required.

- An application monitoring framework is introduced. This framework is an extension of the Application Response Measurement (ARM) standard [OpenGroup01, Johnson00] that provides a method of monitoring the performance of applications within a distributed environment, as well as recording low-level profiling information with minimal overhead. The performance of applications that have been characterised and associated with performance models are monitored using ARM during the application's execution. This measured performance is then compared with the predicted performance to locate any inaccuracies in either the application's characterisation or hardware benchmark measurements. The performance model is then automatically refined in an attempt to correct these inaccuracies, such that future evaluations result in more accurate predictions.

- The ability to characterise and predict the performance of Java applications is implemented. Java's 'compile-once run-anywhere' cross-platform features are a key advantage within heterogeneous environments and, with Java's major performance improvements from next generation runtime optimisation technologies [Sun02a], are a major factor behind Java's recent acceptance into the high performance community [Getov01, Kielmann01]. Java bytecode parsing and instrumentation tools also provide the ability for characterisation at the object level, removing any model inaccuracies that were inherent from compiler optimisations, as well as the requirement of having access to the original source code.

- Instead of a single hardware object, a common platform interface is introduced.

This provides access during a predictive evaluation to a number of performance objects that characterise the performance-critical elements of the application's execution environment. Included with a platform characterisation is an implementation of this interface that models these elements in order to evaluate dynamic performance variations during the application's execution. For example, a Hotspot JVM implementation of this interface is described within this thesis, that models the runtime optimisations of the JVM in order to evaluate accurate predictions of Java applications.

- The ability to assign a confidence metric to all evaluated transactions is realised, facilitating the possibility of a suggested level of accuracy for all predictive results. Evaluations of transactions from historical data, obtained from the application monitoring framework, can be assigned a confidence related to the amount of data currently available regarding the transaction's previous executions. An interface to this historical data is also presented such that more sophisticated confidence calculations can be established if required. Using this confidence metric insures that inaccuracies within a less detailed model can be observed and accounted for by either the developer, or the middleware service performing the evaluation.

1.4 Thesis Structure

This thesis is divided into eleven chapters.

Chapter 2 reviews existing tools and frameworks for the performance modelling and evaluation of distributed applications. Furthermore, PACE is discussed in greater detail, covering the main features of predicting performance using PACE, including a performance characterisation language, hardware benchmarking analysis and an evaluation engine.

Chapter 3 describes the details involved in creating and evaluating a perfor-

mance model using PACE for a given application. The performance of Spatial-Motion Compression (SMC) [Lopez-Hernandez03], a lossless compression algorithm, is characterised, with efforts made to predict data-dependent areas of code. The final model is shown to predict the execution time of SMC within 30% for varying compression parameters and input video streams. Finally, an application steering methodology (introduced in Section 1.1) is implemented, where SMC can use predictive results from the PACE performance model to 'steer' its performance during execution. It is shown that this application steering methodology can be used to achieve optimum compression results given an environmental execution time constraint.

Chapter 4 describes the design of a number of extensions to PACE in order to cope with the dynamic nature of Grid architectures, as outlined in Section 1.3. The new predictive and monitoring framework that results from these extensions is known as 'jPACE' throughout this thesis in order to distinguish it from the original PACE framework. These extensions are supported by the experiences of characterising and predicting the SMC algorithm in Chapter 3.

Chapter 5 introduces the jPACE Performance Characterisation Language (jPCL), a newly developed, flexible characterisation language that is used to describe the performance of distributed Java applications. Each type of performance object defined in jPCL, which is used when characterising the performance of a distributed application, is described, with the construction of the performance model of a simple case study documented for clarity. A tool that has been implemented to automate the characterisation of jPCL transactions and dramatically reduce the time required to create a performance model is also described.

Chapter 6 describes the performance characterisation within jPCL of the application's underlying platform. The platform interface is defined, and the creation of a Hotspot Java Virtual Machine platform implementation, for the predictive evaluation of Java applications, is documented. Each type of performance object that describes the performance-critical elements of the platform is described, as well as a number of

tools used to populate these objects with accurate platform benchmark timings.

Chapter 7 describes the implementation of a parametric evaluation engine, used to evaluate both application and platform characterisations in order to obtain predictive results. This chapter documents how each element of an application's characterisation is evaluated, as well as how both an evaluated response time and associated confidence are assigned to all predictions.

Chapter 8 documents the predictive results obtained from the implementation of these extensions with the performance characterisation and evaluation of three scientific kernels. Each kernel is described, characterised within jPCL and evaluated. It is shown that predictions within 20% of the actual execution time can be achieved with the jPACE environment.

Chapter 9 describes the implementation of an application monitoring framework for distributed applications. The Application Response Measurement (ARM) standard is used to monitor the response time of transactions, and an extension to ARM is documented that facilitates low-level, low-overhead profiling of Java applications. It is then shown how the performance data obtained by this framework during an application's execution can be used to automatically refine jPCL performance characterisations, which result in more accurate evaluations in the future.

Chapter 10 documents the use of jPACE within a number of middleware services to enhance an environment's efficiency. Two environments are used as case studies: the TITAN resource scheduler and a service routing algorithm within an e-business, web services application. It is shown that using performance-based decisions within these two types of distributed architecture can greatly improve the architecture's efficiency and overall performance.

Chapter 11 concludes this thesis, and proposes future work that could enhance the jPACE predictive and monitoring framework.

Chapter 2

The Performance Evaluation of Distributed Applications

The ability to evaluate the performance of an application is vital for achieving the optimum performance for both applications and computer systems in general. During the software lifecycle, it is important for developers to take account of performance, so that performance-critical elements of their application can be located, analysed and optimised during both the design and implementation stages. Furthermore, once the application has been completed, evaluating its performance enables the application to be efficiently allocated within a computational environment.

A number of methodologies have been developed in order to evaluate the performance of distributed applications throughout the software lifecycle, and a wide range of tools have implemented these methods in order to provide a developer or environment with a wealth of both predicted and measured performance data. Each of these tools can be categorised into one of four specific groups: benchmarking, simulation, analytical modelling and monitoring; with the type of tool used generally dependent upon the stage of the application's lifecycle (see Figure 2.1 [Jain91, Smith90]). However, one of the fundamental laws in performance studies is that the performance expert should use more than one of these techniques in order to validate their accuracy [Jain91], such

as monitoring the application during execution in order to determine the accuracy of its prediction.

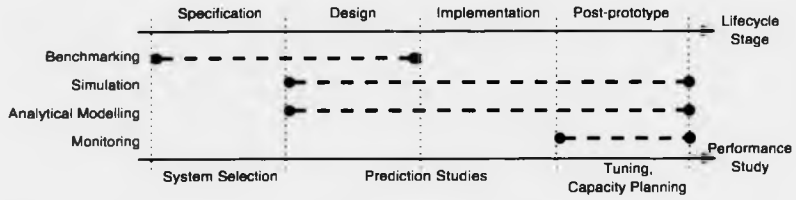


Figure 2.1: The application of performance evaluation methodologies during the software lifecycle. The performance studies generally performed during this lifecycle are: system selection, where the parallel platform that the application will eventually execute upon is chosen; prediction studies, where the performance of the application is studied in order to locate areas of the design that may impede the eventual performance; tuning studies, where performance-critical areas of the application's implementation are optimised; capacity planning studies, where it is ensured that adequate resources are available for future workload demands, while still meeting performance objectives.

This chapter describes these four main performance evaluation categories, and documents a selection of the principle research projects in this area that focus on using these evaluation techniques for the performance prediction of distributed applications. PACE is documented in greater detail here, as it is the experiences of using this framework that this thesis builds upon for the prediction and monitoring of distributed Java applications.

2.1 Benchmarking

Benchmarks can be viewed as test workloads that run on target platforms to measure the performance of system components [Jain91]. The levels of workload represented by benchmarks can range from simple mathematical operations and communication protocols to entire distributed applications. Rather than developing a single benchmark to measure a single workload on a single architecture, benchmarks tend to be developed in suites, representing multiple workloads that characterise a set of similar computational functionality. Each benchmark is normally executed on a range of

differently-performing platforms and execution environments, in order to facilitate a performance-based comparison of these workloads on different architectures.

Benchmarking suites including a hierarchy of benchmarks that attempt to identify the performance of varying aspects of a computing system include [Addison93, Hockney94, vanderSteen93]. The Standard Performance Evaluation Corporation (SPEC) [Dixit93] have developed a wide range of benchmarks for CPU characterisation, as well as workloads for the majority of modern high-performance applications, including Java JVM workloads (in order to compare the performance of different JVMs running on different platforms), client/server workloads, and even mail server benchmarks. A large number of benchmarks are implemented to measure the performance of a range of mathematical kernels, in order to facilitate comparison between these kernels' algorithms and the performance of mathematical operations on a range of platforms. These include, most notably, the Linpack benchmarks for basic algebra computations [Dongarra93] and the NAS parallel benchmarks [Bailey91]. These benchmarks have been ported to the Java platform in order to benchmark the performance of MPI-based mathematical kernels on the JVM, including the JavaGrande [Bull00] and their associated benchmarks [Mathew99].

2.1.1 NAS Grid Benchmarks

With the recent research into Grid architectures, a number of Grid benchmarks are being developed; an example of which is the NAS Grid benchmarks [Wijngaart02]. These benchmarks are currently a 'paper-and-pencil' specification, based on the NAS parallel benchmarks, which propose a set of tasks that assess Grid performance at the user application level. This specification serves as a uniform tool for testing the functionality and efficiency of Grid environments, with users free to implement this specification in order to evaluate the performance of their Grid protocols and user-level applications. For example, such an implementation could be used to compare the performance of the several prototype grid tools that currently exist (Globus, Legion

[Leg99], CORBA [Ben-Naten95], the Sun Grid Engine [Sun00], Condor, TITAN) in order to conclude which Grid middleware implementation would result in the optimum efficiency and provide the greatest performance for a given Grid application.

The NAS Grid benchmarks currently include four families of benchmarks that specify common performance-critical problems within Grid environments. They aim to capture the capabilities of Grids for performance-distributed computations and for accessing data that may reside anywhere across a hierarchy of storage devices, ranging from local disk to remote archival storage. Each benchmark consists of a slightly modified version of NAS parallel benchmark specifications, and is defined by class (mesh size, number of iterations), source(s) of input data, and consumer(s) of solution values. They include:

1. *Embarrassingly Distributed:* which represent the important class of grid applications called parameter studies. These constitute multiple independent runs of the same program, but with different input parameters.

2. *Helical Chain:* which represent long chains of repeating processes, such as a set of flow computations that are executed consecutively, as is customary when breaking up long-running simulations into a series of tasks.

3. *Visualisation Pipe:* which represent chains of compound processes, like those encountered when visualizing flow solutions as the simulation progresses.

4. *Mixed Bag:* which, like the visualisation pipe specification, involve the sequence of flow computation, post-processing and visualisation, but with an emphasis on introducing asymmetry. Different quantities of data are transfered between different tasks, and some tasks require more work than others. This benchmark is therefore useful for testing the efficiency of grid scheduling services.

The eventual goal is to include within the NAS Grid Benchmark suite at least one reference implementation of each of these grid benchmark families, as well as a spec-

ification for a user or developer to construct their own data flow graphs. These graphs would be used as Grid benchmarks, allowing users to augment these benchmarks with their own applications.

2.2 Simulation

Simulation involves the modelling of a system, which is then processed in order to evaluate its performance. Simulators are most useful for evaluating the performance of computing environments that are impossible or too costly to implement or construct. However, simulation models themselves are often time-consuming and costly to run, and tend to produce very large amounts of performance data which can be difficult to analyse.

There are a number of simulation frameworks that have been developed in order to simulate the performance of distributed systems, including the Rice Parallel Processing Testbed (RPPT) [Covington88, Covington91], the Performance Analysis Simulation Environment (PASE) [Papaefstathiou88, Pombortsis94], CHAOS [Uysal98], Parsec [Bagrodia98] and 'à la carte' [Berkbigler03]. Simulation languages have also been developed (ModSim [Herring90] and SimScript [Rusell83] for example), allowing the performance of a computing system to be described within the language and, when evaluated, provides simulated results of that system. Simulation frameworks specifically for Grid computing currently include MicroGrid [Song00], Bricks [Aida00], SimGrid [Casanova01] and GridSim [Buyya02]. À la carte and GridSim, two of the more popular frameworks listed above, are discussed in more detail below.

2.2.1 À la carte

À la carte is a simulation framework for massively-parallel architectures, developed at the Los Alamos National Laboratory. This work focuses on the performance-evaluation of ASCI [Hodges02] codes, which require unprecedented computing power and re-

sources. These applications are facilitated by their deployment on massive computing platforms, consisting of tens of thousands of processors capable of achieving 10-100 TeraOPS, with WAN connectivity from distant sites for remote execution and observation. It is therefore beneficial to simulate the performance of these applications upon perspective hardware prior to purchasing, due to the sheer cost of the computer environments.

The approach used for performance simulation relies on an iterative development process for constructing components of appropriate performance-critical elements within the parallel application. These components are then integrated into a portable and efficient parallel discrete event simulation that is scalable to thousands of simulated computational nodes. Each component may describe processors, switches, network interfaces, or application workloads. Studies of an application upon a particular architecture are achieved by populating the simulation framework with the appropriate components that describes the architecture. Each component is specified in the Domain Modelling Language (DML) [Cowie99], and the handling of discrete events within the simulation is performed by the Dartmouth Scalable Simulation Framework (DaSSF) [Liu02, Nicol02].

To gain as great a knowledge as possible of the application's execution, all simulation output is recorded. However, due to the voluminous nature of this data, a number of filter capabilities are implemented in order to concentrate on the required performance elements. A visualisation tool for this data that renders 3D environments is currently being developed [Berkbigler03], that facilitates an easier-to-digest representation of the gathered performance data.

2.2.2 GridSim

GridSim is a modelling and simulation framework for the performance evaluation of Grid environments. Grids can theoretically involve millions of heterogeneous resources scattered among multiple organisations and administrative domains, and in-

clude large variety of application and user demands. Predicting the efficiency of the various types of Grid resource allocation algorithms available within these environments is therefore important in order to gain an understanding of which of these middleware services should be chosen, given a range of situations, applications, and so on. GridSim has been implemented to obtain these performance insights into these environments.

GridSim is implemented as an object-oriented toolkit for the Java platform on top of SimJava [Howell98], a process-based discrete event simulation package. GridSim simulates time- and space-shared resources within different capabilities, time zones and configurations, and models resource allocation algorithms among these simulated resources in order to obtain an efficiency of Grid schedulers. At the time of writing, a Nimrod-G [Buyya00] economic Grid resource broker simulator has been developed that evaluates scheduling algorithms based on deadline and budget-based constraints. This simulation was used in order to obtain the performance and scalability of a number of scheduling policies with different Grid configurations, such as a varying number of resources, capability, cost and users.

2.3 Analytical Modelling

Analytical modelling is similar to simulation but, due to its small evaluation requirements and greater flexibility, its approach provides more elegant alternatives to performance evaluation. This presents a great advantage over simulation for the performance evaluation of distributed applications, since computation and communication upon and among thousands of platforms, while being very time-consuming to simulate, could be analytically evaluated quickly and efficiently. Analytical techniques include heuristics, assumptions and simplifications in order to further increase their efficiency and, while these simplifications can produce inherent inaccuracies in the evaluated results, hybrid techniques can be implemented that combine analytical methods with benchmarking

and monitoring in order to reduce this inaccuracy. PACE, as well as the performance framework documented within this thesis as an extension to PACE, are examples of this approach.

A multitude of analytical modelling techniques have been developed to evaluate the performance of both the hardware and software of distributed applications and environments. In addition to PACE, the most notable of these include the Performance Oriented End-to-end Modelling System (POEMS) [Deelman98] and SCALability Analyser (SCALA) [Sun02b]. Both of these frameworks are documented below.

2.3.1 POEMS

Similar in design to PACE, POEMS is a modelling framework for the performance evaluation and prediction of distributed applications. Performance-critical elements of the application and its execution environment are described as individual models that span three domains: application, operating system and hardware. The application domain specifies parallel computation as a dynamic task graph, where nodes represent sequential computation units and edges define dependencies. The operating system domain provides the models for process and memory management, inter-process communication and parallel file systems. The hardware domain provides models for the processor and memory components, where the latter includes models for cache memory as well as shared memory hierarchies. As with PACE, a complete performance model consists of a collection of these objects in order to capture every element of the application.

Each model is written within a formal specification language that includes deterministic task graph analysis and LogP [Culler93] and LoPC [Frank97] models that can characterise performance at a variety of levels of abstraction. Parallel computation is described using a generalized task model and, at the time of writing, a task graph construction tool is being developed where these graphs are created automatically during compilation. Predictive results are achieved by evaluating the performance model

using MPI-SIM [Prakash98], an MPI [Gropp96a, Gropp96b] application simulator.

2.3.2 SCALA

SCALA is a framework designed to evaluate scalability analysis of parallel applications executing on massively-parallel architectures. SCALA performance models are automatically constructed after the first execution of the application. Static and symbolic analysis is produced by a restructuring compiler, and then dynamic data is collected during the application's execution. Typically, the application is executed upon a small subset of the eventual parallel architecture where the application will be deployed. After execution, this dynamic data is interfaced with the static data in order to automatically create a performance model. This model can then be evaluated repeatedly so as to predict the modelled application's scalability performance.

In order to collect the dynamic data, a number of compiler technologies have been implemented that instrument the application with custom debugging information during compilation. The debugging output obtained during execution is used to populate the performance model with predictions of otherwise unpredictable codes (data-dependent for example). This custom instrumentation not only facilitates the automated construction of performance models after a single execution, but also integrates with sophisticated visualisation tools that can highlight performance-bottlenecks within the application to a developer. Furthermore, SCALA is currently being developed for the characterisation and prediction of Grid applications [Sun02b].

2.4 Monitoring

Monitoring techniques are used to measure the performance of a system during the application's execution. Such techniques can either report specific areas of the application, or be completely verbose and measure all aspects of the execution. However, it is important to ensure that the overheads of measuring performance do not incur a

performance-hit on the application or that, if they do, are recognised and appropriately incorporated into the historical data.

The majority of monitoring tools are implemented by instrumenting the application with profiling or debugging information. Most compilers have a debugging option, which if used will enable this debugging information to be written to a predetermined output stream. More configurable monitoring tools will instrument either the application's source code or the compiled object code with specific calls at the required performance-critical elements, and it is these elements that will be measured during execution.

Pablo [DeRose98] and Paradyn [Miller95] are noted here for the monitoring of the performance-critical aspects of distributed computer systems. The Application Response Measurement (ARM) standard is a light-weight API for the monitoring of transaction performance within distributed environments and is used as the basis of the automated refinement of jPACE performance characterisations, documented later in this thesis. Networked Application Logger (or NetLogger) [Gunter00b] is another tool for the end-to-end monitoring and analysis of distributed systems. The Grid Monitoring Architecture (GMA) [Tierney01] is an abstract description of the components needed to build a scalable monitoring system suitable for Grid environments. Examples of monitoring frameworks that have started implementing the GMA include the Network Weather Service (NWS) [Wolski99], AutoPilot [Ribler98], Java Agents for Monitoring and Management (JAMM) [Tierney00] and [Waheed00], a design and implementation of an infrastructure that enables the monitoring of resources, services and applications within a computational Grid. The GMA is documented below.

2.4.1 Grid Monitoring Architecture

Within a Grid environment, it is likely that more than one monitoring system will be used to evaluate the performance of Grid applications and services, as different companies and domains will already have specific preferences and requirements for the type

of data that is useful within their scenario. The Grid Monitoring Architecture has been developed to facilitate inter-operation among these monitoring systems by defining an architecture of monitoring components that specifically addresses the characteristics of Grid platforms. A Grid monitoring system is differentiated from a general monitoring system in that it must be scalable across wide-area networks and encompass a large number of heterogeneous resources. The monitoring system's naming and security mechanisms must also be integrated with other Grid middleware. Furthermore, an essential aspect of a Grid monitoring system is a set of common protocols for messaging, data exchange and management.

The GMA consists of three component types:

1. *Directory Service:* a distributed repository service for the publishing of and searching for performance data. Producers and consumers must publish their existence within the directory service, along with their type, the type of events they provide or consume, security mechanisms and so on. Other producers and consumers can then register with this service in order to retrieve this data if/when required.

2. *Producer:* any component that uses the producer interface to send events to a consumer. The core interaction functions that a producer may implement are registration, subscription with other consumers or directory services, and the ability to locate and notify consumers of events.

3. *Consumer:* any component that uses the consumer interface to receive event data from a producer. The core interaction functions that a consumer may implement are registration, subscription with other producers or directory services, and the ability to accept producer events.

The GMA is designed to handle performance data transmitted as time-stamped performance events. An event is a typed collection of data with a specific structure defined

by a schema that is populated as appropriate during the lifecycle of the application or Grid environment. The implementation of sensors that record this information and when this information is recorded is not specified within the GMA specification and can be tailored as necessary. Performance event data is always sent directly from a producer to a consumer.

2.5 The PACE Framework

The Performance Analysis and Characterisation Environment [Cao00, Kerbyson96, Kerbyson98a, Nudd00] was developed by the High Performance Systems Group at the University of Warwick as a framework for developers without expertise in performance-based studies to evaluate and predict the performance of their applications. A PACE performance model contains a number of analytical models that describe the performance-critical elements of the application's computational and inter-resource performance. These models are automatically created by compiling performance characterisation scripts written in a programmatic language called CHIP^{3}S, which has a similar syntax to C. This language makes it simpler for developers to describe their application's performance and create analytical performance models, without the requirement of a detailed knowledge of performance evaluation.

CHIP^{3}S employs a layered approach to performance characterisation, with each layer characterising a specific element of a parallel application (see Figure 2.2). When developing a performance model, each script is associated with a specific layer within the framework in order to characterise a specific performance-critical element of the application. These scripts implement a defined object interface for each layer, providing a framework to enable the re-usability of performance objects.

Applications are characterised within CHIP^{3}S as a control flow of synchronous micro-blocks of either computational or inter-platform communication. Each block is defined within a parallel template by a '`step`' declaration that states either the source,

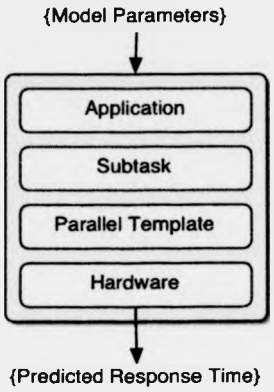


Figure 2.2: A layered methodology for application characterisation including: an application layer for defining global model parameters and the hardware platform that the model will be evaluated on; a subtask layer that characterises the performance-critical sequential computations within the application; a parallel template layer that describes the order of sequential elements and how these elements are spread within a parallel system; a hardware layer that characterises the hardware's computational and inter-resource communication performance.

destination and size of a specific communication type (socket or message-passing, for example) or a reference to a characterised section of computation (declared within the subtask that is associated with this template). This control flow of blocks within a template characterises the 'parallelisation strategy' of the subtask, that is how this computation is spread among the available resources. The complete performance model is a control flow of these subtasks. Each subtask, and in turn each synchronous micro-block, is evaluated as declared within this model control flow.

While the CHIP^{3}S language is prevalent within the PACE framework as a language for the performance characterisation of distributed applications, the PACE framework as a whole is a combination of this language and a number of application and hardware tools. The result is a complete performance evaluation and predictive environment. The PACE toolkit contains: a characterisation tool called '`capp`', which automates the more time-consuming areas of performance model development; a number of hardware benchmarks, in order to accurately obtain timings for a platform's

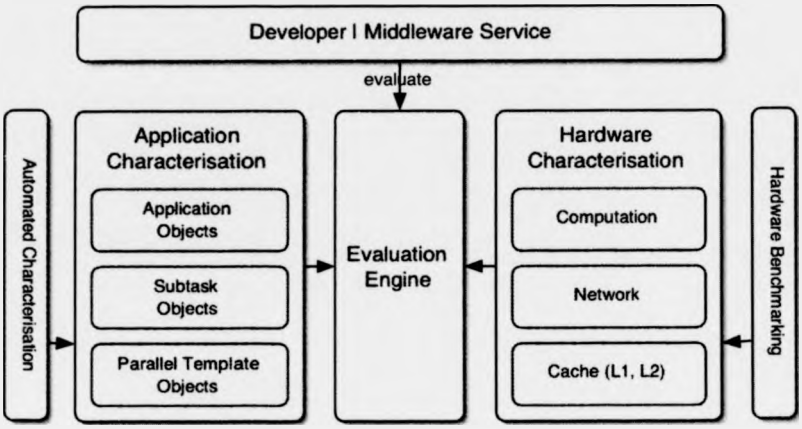


Figure 2.3: The structure of the PACE framework.

computational and communication performance; an analytical methodology for cache performance prediction [Harper99]; an evaluation engine that analytically calculates predictive traces of PACE performance models. An overview of the structure of the PACE toolkit is shown in Figure 2.3.

A detailed description of each of the four layers follows, with an example of a performance object, written in CHIP^{3}S, that is associated within each layer given for clarification. Each object is taken from the characterised performance model of a simple sequential sorting algorithm, the source code of which is shown in Listing 2.1.

2.5.1 Application Objects

A performance model contains one application object that acts as the entry-point to the model's evaluation. Each application object declares the model's parameters, the platform that the model is to be evaluated upon, and the control flow of subtasks within the model. An example application object, taken from the characterised sorting algorithm's performance model, is shown in Listing 2.2.

The '`sort_app`' application object contains a number of '`include`', '`var`',

```
7
8   void sort(int a[], int n) {
9
10      int i, j, temp;
11
12      for (i=0; i<n-1; i++)
13          for (j=n-2; j>=i; j--)
14              if (a[j] > a[j+1])
15                  swap(&a[j], &a[j+1]);
16
17  }
18
19  void swap(int *x, int *y) {
20
21      int temp;
22
23      temp = *x;
24      *x = *y;
25      *y = temp;
26
27  }
28
```

Listing 2.1: A C implementation of a simple algorithm for sorting an initially random integer array 'a' of size 'n'.

```
1
2   application sort_app {
3
4           (* Subtasks, parameters and models to include *)
5           include sort_stask;
6           include hardware;
7           include IntelPII233;
8
9           (* Interface and global variables *)
10          var numeric:
11                  (* Global variables used as model parameters *)
12                  Nelem = 4096;
13
14          (* Linking to other objects *)
15          link {
16                  hardware:
17                          Nproc = 1;
18                  sort_stask:
19                          Nelem = Nelem;
20          }
21
22          (* Options *)
23          option {
24                  hrduse = "IntelPII233";
25          }
26
27          (* Entry point procedure *)
28          proc exec init {
29                  call sort_stask;
30          }
31  }
32
```

Listing 2.2: The application object of the sorting algorithm's PACE performance characterisation.

'link', 'option' and 'proc exec' declarations. 'include' declarations state the name of other performance objects that are referenced within this performance model. In this example these include the 'sort_stask' object (line 5), which is the sorting algorithm's subtask performance object, the 'hardware' object (line 6), which is a generic object that represents characteristics of the hardware during evaluation (such as number of processors), and the 'IntelPII233' object (line 7), which references the hardware object that characterises the performance of an Intel Pentium II 233MHz resource. Including a hardware object does not tell the evaluation engine to evaluate the model on this resource however, this is declared later within the object.

'var' declarations define variables within a performance object. These variables are used to control the flow of evaluation within the model, define elements of performance that can be referenced within a characterisation, define strings for debugging information, and so on. Variables can be declared within any application, subtask or parallel template performance objects. However, any variables declared within the top-level scope of an application object are declared as parameters and can be modified at the start of evaluation. This parametric design of performance models allows data- and processor-scalability analysis to be performed without recompiling the model; the model is instead re-evaluated with a range of processor and parameter initialisations. This example declares one numeric variable called 'Nelem' (line 12) that is initialised to 4096 and used to parameterise the model for a range of elements that are sorted during execution.

'link' declarations allow variables, or references to computation, to be initialised within other performance objects prior to their evaluation. This enables the passing of parameters or calculated expressions among all performance objects within a model in order to appropriately control the evaluation. In this example the 'Nproc' variable within the 'hardware' object is set to 1 (line 17) to indicate a sequential evaluation on one processor, and the 'Nelem' variable within the 'sort_stask' subtask is set to the value of 'Nelem' in the application object (line 19); in this case, if the

parameter is not modified prior to evaluation, 4096.

'option' declarations set a number of options during evaluation regarding the evaluation of specific performance objects. Currently there are three options available within the CHIP^{3}S language [HPSG99]:

1. 'hrduse'. A string value, valid in the application and subtask objects. It controls the hardware model selection and must be defined somewhere within the performance model.

2. 'nstage'. A numerical option that can be set in parallel templates. It sets the number of times the stage is repeated.

3. 'ptmuse'. A string option that can be used in subtask objects to select which parallel template to use.

In this example, the 'hrduse' option (line 24) is set to 'IntelPII233' in order to evaluate this model with the Intel Pentium II 233MHz hardware characterisation.

'proc exec' declarations are procedures that are used to define control flow within performance characterisations. All application, subtask and parallel template objects must have one 'proc exec' declaration called 'init' that is evaluated at the start of the object's evaluation, and can be used either to initialise any variable declarations defined or evaluate other performance objects. The 'init' 'proc exec' declaration within an application object provides the entry-point to the model's entire evaluation. This example declares one 'init' 'proc exec' declaration that evaluates the 'sort_stask' subtask object (line 29).

2.5.2 Subtask Objects

A subtask object characterises an element of sequential computation. Like the application object, the subtask object can define any number of 'include', 'var', 'link'

and '`proc exec`' declarations, as well as '`proc cflow`' declarations that characterise computational performance. Listing 2.3 shows an example subtask object, taken from the characterised sorting algorithm's performance model.

```
1
2   subtask sort_stask {
3
4           (* Subtasks, parameters and models to include *)
5           include async;
6           include hardware;
7
8           (* Interface and global variables *)
9           var numeric:
10                  (* Passed variables from application object *)
11                  Nelem;
12
13          (* Linking to other objects *)
14          link {
15                  (* Link the function to the parallel template *)
16                  async: Tx = sort();
17          }
18
19          (* Entry point procedure *)
20          proc exec init {}
21
22          (* Calls: swap *)
23          proc cflow sort {
24                  compute <is clc, FCAL, 2*FARL, SILL>;
25                  loop (<is clc, LFOR>, Nelem - 1) {
26                          compute <is clc, 2*AILL, CMLL, TILL>;
27                          loop (<is clc, LFOR>, (Nelem - 2) / 2) {
28                                  compute <is clc, 2*CMLL, 2*ARL1>;
29                                  case (<is clc, IFBR>) {
30                                  0.5:
31                                          compute <is clc, 2*ARL1>;
32                                          call cflow swap;
33                                  }
34                                  compute <is clc, INLL>;
35                          }
36                          compute <is clc, INLL>;
37                  }
38          } (* End of sort *)
39
40          (* Calls: *)
41              proc cflow swap {
42                  compute <is clc, FCAL, 2*FARL, 4*POL1, 3*TILL>;
43              } (* End of swap *)
44
45  }
46
```

Listing 2.3: The subtask object of the sorting algorithm's PACE performance characterisation.

This subtask contains two '`include`' declarations (lines 5 and 6) that reference the '`async`' parallel template object[1], to characterise a sequential parallelisation strategy, and the generic '`hardware`' object. A numeric variable called '`Nelem`'

[1] while the name '`async`' may sound like 'asynchronous', it actually characterises one synchronous block of computation, without any communication

(line 11) is declared, whose value is initialised prior to the subtask's evaluation by the application object's '`link`' declaration. Furthermore, an '`init`' '`proc exec`' declaration is included, although this is empty as no initialisation control flow is necessary for this object's evaluation.

The variable '`Tx`' within the '`async`' parallel template object is referenced to the evaluated execution time of the '`sort`' '`proc cflow`' declaration. This declaration is the CHIP^{3}S characterisation of the original '`sort`' method within the algorithm's source code. '`proc cflow`' characterisations can contain any number of four statements that capture the method's performance:

1. '`Compute`': that calculates the execution time of a list of instructions that is given to the statement as parameters. For example, line 34 computes the execution time of the '`clc`' instruction '`INLL`'. To calculate this, the parallel template that is evaluating this '`cflow`' looks up the value associated with the '`INLL`' instruction in the hardware object being used for the current evaluation. This value is then added to the total predicted execution time for the current '`cflow`'. A more complicated list of machine instructions can also be passed to the '`compute`' statement, such as that shown at line 42.

2. '`Loop`'. The '`loop`' statement is a CHIP^{3}S characterisation of an iterative statement ('`for`', '`while`' and so on) that is present in the original application. The loop count of this iterative statement is characterised by the statement's second parameter ('`Nelem - 1`' in the case of the '`loop`' statement at line 25). This variable may be a constant defined previously in the subtask, or an expression that relates to a number of model parameters that have been passed from the model's application object.

3. '`Case`'. The '`case`' statement is a CHIP^{3}S characterisation of a conditional statement ('`if`', '`switch`' and so on) that is present in the original application. This statement can define a number of performance characterisations that are

evaluated according to their probability of execution (0.5 in the '`case`' statement at line 30). This '`case`' statement characterises the probability of the '`if`' statement in the original application and, as the array being sorted is initially random, its value is set to 0.5.

4. '`Call`'. The '`call`' statement evaluates another '`proc cflow`' statement, adding the predicted execution time of that statement to the total predicted execution time for the current '`cflow`'.

'`capp`' is a tool that automates the construction of '`proc cflow`' statements within subtasks by characterising the performance of an application's C source code. Automating these characterisations greatly reduces the time required for PACE performance model development, as well as ensuring that no mistakes are made within these declarations. For this reason, '`capp`' was used in this example to characterise the sorting algorithm's '`sort`' and '`swap`' methods.

2.5.3 Parallel Template Objects

A parallel template object consists of a control flow of a number of synchronous microblocks that characterise the parallelisation strategy of its associated subtask object. Each block can either contain a specific communication paradigm (defined by the source and destination platforms and the size of the communication) or a computation that is evaluated on all the available resources (the performance of which is characterised by a '`proc cflow`' declaration within the subtask). A single microblock is characterised within CHIP^{3}S by a '`step`' declaration. Listing 2.4 shows an example parallel template object.

This example describes the control flow of a number of microblocks that characterise a single computation, followed by a communication of data between neighbouring processors. Each '`step`' declaration characterises a specific type of performance, and the declaration's parameter specifies which type of performance is being

```
1
2   partmp mitmp {
3
4           include hardware;
5
6           var compute: Tx;
7           var numeric: N;
8
9           proc exec init {
10
11                  var numeric: i;
12
13                  step cpu {
14                          confdev Tx;
15                  }
16
17                  for (i = 1; i <= hardware.Nproc - 1; i = i + 1) {
18                          step mi_send {
19                                  confdev i, i + 1, N * N * 8;
20                          }
21                  }
22
23                  for (i = 2; i <= hardware.Nproc; i = i + 1) {
24                          step mi_recv {
25                                  confdev i - 1, i, N * N * 8;
26                          }
27                  }
28
29          }
30  }
31
```

Listing 2.4: An example parallel template object.

stated for this declaration: '`step cpu`' (line 13) characterises a computation; '`step mi_send`' (line 18) characterises a communication; and so on. Within the '`step`' declaration, the specifics of the type of performance are characterised with a '`confdev`' statement. This statement either defines a '`compute`' variable for computation (on line 14 the compute variable '`Tx`' reference is specified, which would have been linked within a subtask object to a '`proc cflow`' declaration) or the source, destination and size of the communication (on line 19 the numeric variable '`N`' is used to specify the communication's size, which would also have been linked from a subtask object). The parallelisation strategy of the majority of parallel applications can be characterised with a control flow of these '`step`' declarations.

The sorting algorithm is sequential and so a simple parallel template that characterises the execution of the algorithm's subtask on all the resources is used within the algorithm's performance model. This parallel template is shown in Listing 2.5.

```
1
2   partmp async {
3
4           include hardware;
5
6           (* Sequential execution time *)
7           var compute: Tx;
8
9           proc exec init {
10
11                  step cpu {
12                          confdev Tx;
13                  }
14
15          }
16  }
17
```

Listing 2.5: The parallel template object of the sorting algorithm's PACE performance characterisation.

2.5.4 Hardware Objects

A hardware object characterises the computational and inter-resource communication performance of the underlying platform. CHIP^{3}S characterises a method's performance as a control flow of machine-code instructions, and the hardware object contains benchmarked timings for each of these instructions. During evaluation, timings for these instructions are located within the specified hardware object and used to calculate the model's predicted performance. It is important to accurately measure these timings if accurate predictive evaluations are to be achieved. An example hardware object, for the '`IntelPII233`' hardware object defined within the algorithm's characterisation, is shown in Listing 2.6.

2.5.5 Evaluation

Once the performance-critical elements of the application and its platform have been characterised, a performance model can be created. This is achieved by compiling each performance object with the CHIP^{3}S compiler and then linking each compiled object together to form a single executable, that can be evaluated to provide a predictive analysis of the characterised application. Instead of compiling each performance object

```
5
6   config IntelPII233 {
7
8           hardware {
9                   Tclk = 1 / 233,
10                  Desc = "PC, Intel PII/233MHz, 64MB, Linux 2.2",
11                  Source = "tango.dcs.warwick.ac.uk";
12          }
13
14          (* C Operation Benchmark Program $Revision: 1.1 $
15             Timer overhead 2.05963000 *)
16
17          clc {
18                  SISL = 0.0119327,
19                  SISG = 0.0176694,
20                  SILL = 0.00481602,
21                  SILG = 0.00482936,
22                  SFSL = 0.00532936,
23                  SFSG = 0.00571936,
24                  SFDL = 0.010006,
25                  SFDG = 0.011056,
26                  SCHL = 0.00552602,
27                  DFSG = 0.153739,
28                  DFDL = 0.149532,
29                  DFDG = 0.148572,
30                  DCHL = 0.157172,
31
32  ...
33
```

Listing 2.6: A portion of the hardware object that characterises the performance of a Pentium II 233MHz processor.

to native code, the CHIP^{3}S compiler converts the characterisation into C source code, which includes an implementation of an analytical model of that object. A standard C compiler ('gcc' in this case) is then used to link each performance object translated into C code, and a standard CHIP^{3}S library, to produce the final executable. This library provides each performance model with a standard implementation and tools for performance analysis and evaluation.

An evaluation can be achieved using two different methods:

1. *On the command line.* Executing the model on the command line without any parameters returns one number, which is the predicted execution time for the characterised application with the default parameter set hard-coded into the application object. These model parameters can be changed by initialising new values as parameters to the executable; executing the model while changing the number

of processors can, for example, facilitate application processor-scalability analysis. If passed to the model, standard PACE parameters can provide more detailed debugging and performance information during an evaluation, such as full predictive traces and communication analysis. This 'on the command line' method is usually used by developers for test purposes while the performance model is being refined.

2. *Remote evaluation.* A predicted execution time can also be achieved from within an application by invoking a number of PACE API calls. During compilation, each model is linked with the standard CHIP^{3}S library, which provides a simple API for remote evaluation and allows applications to evaluate a performance model without running a system command. A `'reval()'` method allows an application to initialise a performance model, set the model parameters as necessary, and then evaluate the model to retrieve a predicted execution time. Remote evaluation is usually used by more complex applications that access performance models within complex environments.

Evaluations typically take less than 1 second to complete and have been shown to have an accuracy of within 10% for a wide range of parallel applications. However, these accuracies are highly dependent upon the model's performance characterisations and the hardware object's benchmarked timings. It is generally the case that the greater the time taken to continuously refine the model from historical data, the greater the eventual accuracy achieved.

2.6 Summary

This chapter documented a selection of the current research in the area of performance evaluation of distributed applications. Techniques for performance evaluation fall under one of four categories (benchmarking, simulation, analytical modelling and monitoring) and this chapter described a number of tools that are associated with each of

these categories. Of particular interest is the growth of performance frameworks for the evaluation of Grid architectures; performance-based services will be highly important within computational Grids in order to efficiently schedule applications of varying priorities among geographically-dispersed administrative domains.

Described in greater detail was the PACE framework. The following chapter documents the experiences of characterising a lossless compression algorithm with PACE and achieving accurate predictive results for a wide range of input video streams and compression parameters.

Chapter 3

Predicting the Performance of Applications using PACE

The previous chapter described a number of tools that provide detailed performance information regarding the execution of distributed applications. Whether this information is predicted prior to the application's execution or monitored and recorded during, such information can be valuable to both developers (to provide focus on performance-hindering areas of an application) and middleware infrastructures (to achieve efficient resource allocation within dynamic computing environments).

A suggested method for use within a middleware infrastructure is application steering. As previously discussed, application steering is a technique for providing the application with the ability to predict its own performance, allowing its parameters to be appropriately chosen to meet a certain restriction. For example, there may be a time constraint on the execution of an application, and by predicting its performance prior to execution, the application can be 'steered' in such a way as to meet that time constraint.

This chapter describes the process of characterising and predicting an example application, the Spatial and Motion Compression (SMC) [Lopez-Hernandez03] algorithm, using the PACE toolkit described in Chapter 2. First the algorithm is charac-

terised using CHIP^{3}S before being compiled into a performance model. This model is then used to predict the algorithm's performance using a number of different input datasets, and the accuracy of these predictions is also measured. Application steering is then implemented to provide the algorithm with the ability to choose the optimum compression parameters prior to its execution such that the algorithm completes within a given time constraint.

This chapter includes:

- A description of the SMC compression algorithm.
- The specifics involved in creating a PACE performance model, including characterising the performance of an application and achieving accurate predictions of data-dependent code.
- The implementation of an application steering methodology using the compression algorithm and the PACE performance model to achieve the optimum compression results possible under a given execution time constraint.
- A summary of the key advantages and disadvantages of using PACE to predict the performance of applications.

3.1 The Spatial and Motion Compression Algorithm

SMC (Spatial and Motion Compression) is a lossless compression algorithm that can be applied to the compression of computer-generated animation sequences. It combines several lossless compression techniques to exploit the spatial and temporal redundancies found in computer animations. Consecutive animation frames are very similar; most of the objects in one frame reappear in the next frame with slightly different positions and orientations. These kinds of similarities or redundancies can be exploited using a temporal compression technique. However, there may also be new

information that cannot be inferred from previous frames. For example, an object that was occluded by another moving object, new information introduced on one side of the image while the camera is panning, new image details appearing when the camera is zooming in, a face that was occluded on a rotating object, and so on. This information can sometimes be better encoded with a spatial compression technique, using neighbouring information from the same frame. In particular, a spatial compression technique exploits the redundancies found in areas filled with the same colour or areas with smooth changes of colour.

This section describes the components of the SMC lossless compression technique for computer animation sequences. Good compression results are achieved by combining spatial and temporal compression as well as a number of other image encoders. Each encoder is parameterised such that higher levels of compression can be achieved at the expense of execution time; if execution time is not abundantly available however, smaller compression ratios (measured at roughly 4:1) resulting in quicker execution can be achieved.

3.1.1 Compression Framework

A schematic view of the SMC compression technique is shown in Figure 3.1. The main part of this technique consists of a loop in which the spatial and temporal compression encoders are applied. This loop is performed for the number of reference frames specified, from 0 (no reference frames) up to n previous frames. The main processing stages contained within SMC are as follows:

1. Initially the current frame (frame i) is encoded using the LOCO [Weinberger98] spatial encoding technique[1].

2. Block movement is calculated for each reference frame up to a specified n previous frames (frames i-1, i-2, i-3, ..., i-n). Each frame is split into a number

[1] while CALIC [Wu97] is the best spatial coder currently available, LOCO uses less prediction rules and is therefore faster

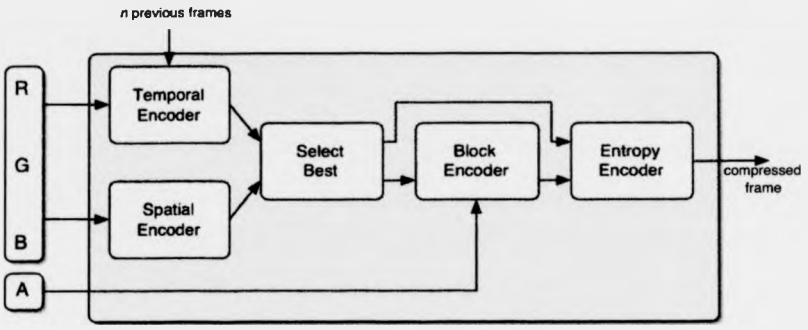


Figure 3.1: The individual components of the SMC algorithm.

of blocks (the size of which is a parameter to the algorithm) and then similar blocks are searched for within a specific search space (also a parameter to the algorithm) in each reference frame. Compression ratios and execution times are highly dependent upon the values of these two parameters. This encoding results in a three-element motion vector for each block: a position movement vector (x, y) relating the most similar block to the original, and a z component indicating the reference frame in which the matching block can be found.

3. For each block in the image, the block best matched to it, either from the spatial prediction or from one of the motion calculated frames, is selected. Then a residual is calculated by the difference between the frame constructed (with spatial and motion blocks) and the original frame. This residual combined with the motion vectors is all the information required to allow lossless reconstruction later. Redundant motion vectors are then removed. These occur since the blocks encoded using the spatial compression technique do not have any motion information.

4. The alpha channel and the vector component z are encoded using a block area encoding [Gilbert98] technique. Due to the vector z's small alphabet (equal to the number of previous frames), large blocks of the same character frequently

arise. Block area encoding replaces these large blocks with a special character followed by the size of the area and its colour. Areas of continuously changing characters are kept intact.

5. Finally the residual and the vector information are further encoded using an entropy coder. There are two options available in this case: the fastest is a rice entropy encoder [Rice79]; the second is a rice entropy encoder followed by an arithmetic encoder.

3.1.2 SMC Compression Parameters

There are several parameters that affect the final compression ratio and execution time [Lopez-Hernandez03]:

1. *Search space:* The size of the search space used in block matching. The execution time usually increases to the square of the search space. The compression ratio is usually better with bigger search spaces.

2. *Block size:* The size of the blocks used in block matching and in the selection of the best technique (spatial or temporal). This is typically set to 2, as this has been derived empirically within this research as usually giving the best compression ratio. Generally, the block size does not affect the execution time.

3. *Previous frames:* The number of previous frames used in temporal compression. The execution time usually increases linearly with the number of previous frames; the compression ratio can increase using more previous frames. A value of either 1 or 3 frames has been derived empirically as a suitable value for this parameter in order to obtain a good compression ratio within a reasonable execution time.

4. *Applying arithmetic coding to the residual:* This is an option that determines whether to apply the arithmetic coder during compression. The rice encoder is

always used to encode the residual, regardless of this option. Arithmetic coding is slower but produces better compression ratios.

These parameters can either be set by the developer prior to execution or automatically set under the application steering methodology described in the later sections of this chapter. However, before application steering can be implemented for any application, the application must first be able to predict its own performance prior to execution, with the use of a PACE performance model, for example. The lifecycle of creating a PACE performance model for the SMC algorithm, from characterising the performance of the algorithm, to creating the performance model itself, is detailed in the following sections.

3.2 Characterising the SMC Algorithm

As documented in Chapter 2, predicting the performance of an application within the PACE predictive framework is only possible once the performance of the application has been characterised. This characterisation is achieved by describing the main sequential elements of the application as well as the communication between them in CHIP^{3}S, a performance-based characterisation language that is part of the PACE framework. Once each performance-critical element of the application has been described within CHIP^{3}S, the characterisation can be compiled into a single PACE performance model that can be evaluated to achieve predictions.

To characterise an application, not only must the original source code be accessible, but the model developer must also have considerable knowledge of the application. Sequential areas of code are characterised in CHIP^{3}S by describing each individual machine-code instruction for every method described within the model. This process can be automated using '`capp`', however the source code must be present for this tool to be used. Furthermore, while determining the performance-critical areas of an application can be reasonably straight-forward, repetitive and decision-making

areas within sequential and communication codes are characterised according to loop count and probability variables, most of which are defined within an expression containing the application's parameters. These variables can be very difficult to resolve without a detailed knowledge of the application in question.

Performance-critical elements within a performance model consist of at least one characterised sequential object running on at least one hardware object. For a parallel application where sequential elements are spread among a number of processors, the communication between sequential areas of code is also characterised. Using a standard text editor, each object is written in the CHIP^{3}S language, with the aid of a number of tools, such as '`capp`', to increase the speed of their creation. Each model also consists of an entry point (the application object), which is the first object to be evaluated during a prediction. The details involved in creating these objects for the SMC algorithm follow.

3.2.1 Sequential Elements: Subtask Objects

Performance-critical sequential elements within an application are characterised in CHIP^{3}S by a number of subtask objects. Each object describes the original sequential source code of the sequential element as well as the relation between parameters used within this object and those globally defined by the application object. The five encoders used within the SMC algorithm, 'spatial', 'temporal', 'block', 'rice' and 'arithmetic', were each characterised within their own subtask object and their individual encoder-specific parameters and source code implementation characterised within each.

Part of the characterisation of the spatial encoder subtask can be seen in Listings 3.1 and 3.2. The subtask defines its associated parallel template at line 6, and then links the entry '`cflow`' statement to that parallel template at line 19. This enables the parallel template to evaluate the subtask by evaluating this '`cflow`' statement, the characterisation of the original method named '`SMC_gap_encoder`' in this example.

Parameters passed from the application object are defined at line 11. The loop count and probability variables that define the control flow of the '`cflow`' statements are first declared at lines 13-14, and initialised in the '`init`' '`proc exec`' statement at lines 23-31.

```
1
2   (* Spatial Encoder Subtask *)
3   subtask spatial {
4
5     (* Subtasks, parameters and models to include *)
6       include async;      (* Asynchronous parallel template *)
7
8     (* Interface and global variables *)
9       var numeric:
10        (* Passed variables from application object *)
11        Wframe, Hframe,
12        (* Probobility and LoopCount definitions for cflow structure *)
13        PneighbourW, PneighbourN, PneighbourNW, PgapCWgeCN,
14        PgapCNWgeMAX, PgapCNWleMIN, LCgap_encoder1, LCgap_encoder2;
15
16    (* Linking to other objects *)
17      link {
18        (* Link the function time to the parallel template *)
19        async: Tx = SMC_gap_encoder();
20      }
21
22    (* Entry point procedure *)
23      proc exec init {
24        PneighbourW = 1 / Wframe;
25        PneighbourN = 1 / Hframe;
26        PneighbourNW = PneighbourW + PneighbourN;
27        PgapCWgeCN = 0.65;
28        PgapCNWgeMAX = 0.53;
29        PgapCNWleMIN = 0.67;
30        LCgap_encoder1 = Wframe * Hframe;
31        LCgap_encoder2 = 2;
32      }
33
```

Listing 3.1: The Spatial Encoder subtask.

The '`SMC_gap_encoder`' '`proc cflow`' statement from this spatial encode subtask is shown in Listing 3.2, as well as the method's original source code in Listing 3.3. This '`cflow`' is the output given from '`capp`' after the original source code of the method '`SMC_gap_encoder`' was characterised.

Examining this '`proc cflow`' gives a better understanding of how important the model's parameters and their related variables are to the subtask's evaluation and the predicted execution time. The '`LCgap_encoder1`' loop count variable from Listing 3.2 is defined in Listing 3.1 as the multiplication of the application parameters

```
87
88      (* Calls: SMC_gap *)
89      proc cflow SMC_gap_encoder {        (* Defined at spatial.c:131 *)
90          compute <is clc, FCAL, 8*ARL1, 5*POL1, AILL, 3*TILL>;
91          loop (<is clc, LFOR>, LCgap_encoder1) {
92              compute <is clc, CMLL>;
93              call cflow SMC_gap;
94              compute <is clc, 2*POL1, AILL, TILL, 2*INLL, SILL>;
95              loop (<is clc, LFOR>, LCgap_encoder2) {
96                  compute <is clc, CMLL>;
97                  call cflow SMC_gap;
98                  compute <is clc, POL1, AILL, TILL, 3*INLL>;
99              }
100         }
101     } (* End of SMC_gap_encoder *)
102
```

Listing 3.2: The '`SMC_gap_encoder`' '`proc cflow`' statement from the Spatial Encoder subtask.

'`Wframe`' and '`Hframe`' at line 30. Both these parameters can be changed at the start of evaluation such that the prediction of a video stream containing varying image sizes can be predicted. Varying values of both '`Wframe`' and '`Hframe` would provide vastly different predicted execution times due to the number of times that the '`loop`' statement is processed. This would be expected, since the real execution time under these conditions would also vary. However, if any of these parameters are inaccurate, inaccuracies in the predicted execution time will be inevitable, and it is therefore important that measures are taken to ensure the accuracy of these parameters for all '`loop`' and '`case`' statements.

```
129
130 void SMC_gap_encoder (SMC_workspace *ws)
131 {
132   SMC_Mem_t *op, *pp, *pp_end;
133   SMC_Mem_t old_res;
134   int c;
135
136   pp_end = ws -> pf.mem + ws -> pf.default_size;
137   for (op = ws -> root -> frame, pp = ws -> pf.mem; pp < pp_end; ) {
138     SMC_gap (op, pp, ws);
139     old_res = *op - *pp;
140     for (++op, ++pp, c=1; c < MAX_COLOUR; ++c, ++pp, ++op) {
141       SMC_gap (op, pp, ws);
142       *pp += old_res;
143     }
144   }
145 }
146
```

Listing 3.3: The original source code of the '`SMC_gap_encoder`' method.

3.2.2 Communication Analysis: Parallel Template Objects

Each subtask within an application is associated with a single parallel template that defines which processors the subtask is concurrently executed upon and characterises the size and type of communication between these processors. However, since the SMC algorithm used is purely sequential, no inter-processor communication elements are involved.

Embedded within CHIP^{2}S is a sequential parallel template called '`async`' that, when evaluated, returns the predicted time for the sequential element associated with it. The listing for the '`async`' parallel template can be seen in Listing 3.4. As there is no communication involved among the five encoders within the SMC algorithm, all five subtasks were associated with the 'async' parallel template.

```
1
2   (* async.la - Sequential 'parallel' template *)
3   partmp async {
4
5     include hardware;
6     var compute: Tx;          (* Sequential execution time *)
7
8     option {
9       nstage = 1, seval   = 0;
10    }
11
12    proc exec init {
13      step cpu {
14        confdev Tx;
15      }
16    }
17  }
18
```

Listing 3.4: The '`async`' parallel template.

3.2.3 Model Entry-point: Application Object

The application object within a performance model includes: a list of model objects used within the entire performance model; initialises the main parameters to the model that can be changed at the beginning of an evaluation, as well as other global variables within the model; sets the hardware resource that the model is to be evaluated on; de-

scribes the control flow of how the sequential objects are evaluated during a prediction (defined within the 'init' 'proc exec' statement). The result of evaluating this 'init' 'proc exec' statement is the total predicted execution time for the entire performance model. The application object that characterised the SMC algorithm is shown in Listings 3.5, 3.6, 3.7 and 3.8.

A list of other objects found within the model can be seen at lines 6-13 of Listing 3.5. These include the subtasks and parallel template described above, the generic 'hardware' object and a 'SunUltra5_360' object (the resource benchmark for a SunUltra V with a 360MHz UltraSparc II processor). Line 17 then defines this 'SunUltra5_360' object as the hardware resource to use for all hardware computation measurements throughout an evaluation.

```
1
2  (* SMC Algorithm *)
3  application smc_compress {
4
5    (* Subtasks, parameters and models to include *)
6    include spatial;        (* Spatial Encoding subtask *)
7    include temporal;       (* Temporal Encoding subtask *)
8    include block;          (* Block Area Encoding subtask *)
9    include arithmetic;     (* Arithmatic Encoding subtask *)
10   include rice;           (* Rice Arithmetic Encoding subtask *)
11   include async;          (* Asynchronous parallel template *)
12   include hardware;       (* Hardware parameters *)
13   include SunUltra5_360; (* This computer's hardware model *)
14
15   (* Options *)
16   option {
17     hrduse = "SunUltra5_360";
18   }
19
```

Listing 3.5: The SMC application object 'include' and 'option' statements.

Parameters to the model are defined and initialised at lines 23-32 of Listing 3.6, and are based on the input parameters of the original SMC algorithm. The values of these parameters can be changed at the start of an evaluation so as to predict an execution of the algorithm with, for example, different types of video streams (parameters 'Wframe' and 'HFrame' for the width and height respectively of every frame) or different compression parameters (parameter 'Nprevframe' for the number of previous frames compared during a temporal sweep). Any parameters that are not changed at the

start of an evaluation are set with their default value as defined within the application object.

One parameter not defined by the SMC algorithm is the '`Pprevframe`' parameter on line 26. This parameter is used as a way of detecting simplicities in the video stream being compressed, which result in varying execution times. This is discussed in more detail later in this section.

```
19
20    (* Interface and global variables *)
21    var numeric:
22    (* Global variables used as model parameters *)
23      Nframe = 20;          (* Number of frames *)
24      Nchannel = 4,         (* Number of channels [RGB] [RGBA] *)
25      Nprevframe = 3,       (* Number of previous frames for encoding *)
26      Pprevframe = 1,       (* Probability of spatial encoding *)
27      Wframe = 720,         (* Width of frame *)
28      Wblock = 2,           (* Width of block *)
29      Hframe = 432,         (* Height of frame *)
30      Hblock = 2,           (* Height of block *)
31      Sspace = 2,           (* Search space for temporal encoding *)
32      aFlag = 1,            (* Arithmetic flag for rice encoding *)
33
34      (* Global variables passed to the various subtasks *)
35      Npfpassed, Wbfpassed, Hbfpassed, Nsspassed, Nsricepassed;
36
```

Listing 3.6: The SMC application object '`var numeric`' statement.

Global variables within the model (defined at line 35) are used to link parameters both to other model objects and within the '`init`' '`proc exec`' statement. These variables are not initialised on declaration and cannot be modified at the start of evaluation in the same way as model parameters. The parameters and global variables required by other objects within the model are passed to these objects within the '`link`' section at lines 39-51 (Listing 3.7).

The '`init`' '`proc exec`' statement can be seen in Listing 3.8. It consists of a '`for`' loop which iterates for the number of frames in the video stream ('`Nframe`'), evaluating all five SMC encoders for that frame in the correct order, as outlined within the SMC description. The values of the model parameters '`aFlag`' (execute the arithmetic encoder on the result of the rice encoder) and '`Nchannel`' (the number of channels within the input video stream) determine the number of times the '`block`' and

```
36
37    (* Linking to other objects *)
38    link {
39      hardware:
40        Nproc = 1;
41      spatial:
42        Wframe = Wframe, Hframe = Hframe;
43      temporal:
44        Nprevframe = Npfpassed, Pprevframe = Pprevframe, Wframe = Wframe,
45        Wblock = Wblock, Hframe = Hframe, Hblock = Hblock, Sspace = Sspace;
46      block:
47        Wblockframe = Wbfpassed, Hblockframe = Hbfpassed;
48      arithmetic:
49        NSymbolSize = Nsspassed, Pprevframe = Pprevframe;
50      rice:
51        NSymbolSize = Nsricepassed;
52    }
53
```

Listing 3.7: The SMC application object '`link`' statement.

'`arithmetic`' encoders are evaluated during a prediction; the '`arithmetic`' encoder is executed for every channel within the video stream and is therefore evaluated a fourth time if an alpha channel is present. Any global variables that were defined at line 35 are set with the required values prior to evaluating a subtask that uses them. The execution time calculated after evaluating this '`proc exec`' statement is the predicted execution time for the SMC algorithm when using the specific compression parameters for that particular evaluation.

3.2.4 Hardware Benchmarking: Hardware Object

The predicted execution time retrieved from a performance model evaluation is in essence the accumulation of all the atomic computation and communication measurements described within the subtask and parallel template objects. These measurements are retrieved from the hardware object defined within the application object during an evaluation, and so it is very important that these measurements are accurate if an accurate prediction of the application concerned is to result.

PACE includes a tool that automatically creates a hardware object for a given resource by benchmarking that resource and measuring the average time it takes to execute each machine-code instruction. Inter-resource communication is also mea-

```
53
54    (* Entry point procedure *)
55    proc exec init {
56      var numeric: i;
57
58      for (i = 0; i < Nframe; i = i + 1) {
59        call spatial;
60        if (i <= Nprevframe) { Npfpassed = i; }
61        call temporal;
62        Nssricepassed = 3 * Wframe * Hframe;
63        call rice;
64        if (aFlag != 0) {
65          Nsspassed = 3 * Wframe * Hframe / 6;
66          call arithmetic;
67        }
68        if (Nprevframe != 0) { if (i > 0) {
69          Nsspassed = (Wframe * Hframe) / (Wblock * Hblock);
70          call arithmetic;
71          call arithmetic;
72          Wbfpassed = Wframe / Wblock;
73          Hbfpassed = Hframe / Hblock;
74          call block;
75          call arithmetic;
76        }}
77        if (Nchannel == 4) {
78          Nsspassed = Wframe * Hframe;
79          Wbfpassed = Wframe;
80          Hbfpassed = Hframe;
81          call block;
82          call arithmetic;
83        }
84      }
85    }
86  }
87
```

Listing 3.8: The SMC application object '`init`' '`proc exec`' statement.

sured by running and measuring the average communication time of a number of standard communication API calls. Each measurement is then written to the hardware object such that they can be retrieved during an evaluation. A hardware model for the '`SunUltra5_360`', one of the available architectures, was benchmarked and included within this case study.

For accurate predictions to be achieved (assuming the application characterisation itself is accurate) the atomic measurements listed within the hardware object must be as close to as possible, if not equal to, the actual atomic computation and communication execution times achieved while the application is executing. Since the real execution times achieved however are highly dependent upon the current state of the machine (CPU load, memory usage and so on) during the application's execution, any differences in the state of the machine during the hardware benchmark and while the

application is executing will provide inaccuracies in the performance prediction.

To overcome this, the atomic hardware measurements were obtained while the resource was under as little load as possible; no other applications were running during the benchmark. This means that the predictions were made on the assumption that the SMC algorithm was the sole application executing upon that resource.

The final Hierarchical Layered Framework Diagram (HLFD) for the characterised SMC algorithm is shown in Figure 3.2.

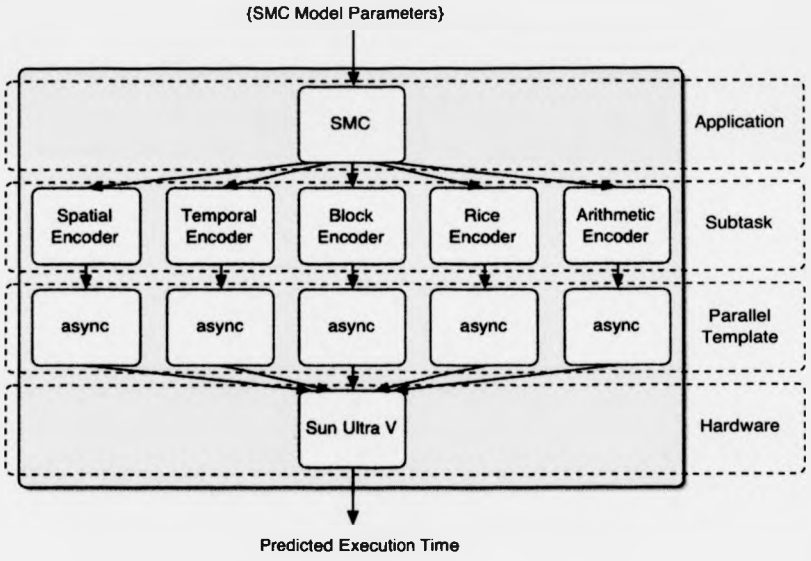


Figure 3.2: The final characterised SMC HLFD diagram, consisting of one application object, five sequential subtask objects with their associated parallel template objects, and one hardware object.

3.2.5 Model Refinement and Data Dependency

During the characterisation of the SMC algorithm, it was made clear that the accuracy of the predictions achieved depends significantly upon the values of the parameters and variables of each performance object. These values within a performance model fall

into three distinct categories:

1. *Constant:* The value of the variable or parameter is always a constant, and is not related to any other parameters or the application's input data set.

2. *Parameter Dependent:* The value of the variable or parameter is related to the values of the input parameters to the application, and can always be calculated accurately from these values.

3. *Data Dependent:* The value of the variable or parameter is related to the application's input data set, and can therefore not be calculated without some prior knowledge of the data set being used.

As a compression algorithm, the SMC performance model had a number of significant variables that were data dependent. While the author of the algorithm was able to provide valuable insight into the correct values of these variables, it was important to profile the application such that these data dependent values could be confirmed after a number of executions[2]. Comparing the values already set for these variables with the debugging information achieved from a number of algorithm executions, the model could be further refined to produce accurate predictions for the set of data used during profiling.

However, changing the input data to a video stream of significantly different complexity produced a vast change in the algorithm's execution time. A main part of the algorithm's execution consists of searching for similarities between the current frame and previous frames; if a simple video stream with many similar successive frames is compressed, the time taken for the algorithm to find these similarities is greatly reduced. Since the model was characterised using profiling information from the original video stream, the model was highly inaccurate for any video streams different in complexity from the original.

[2]profiling was achieved by setting the compiler to full debugging mode and using 'gcov' to list the debugging information regarding the previous execution

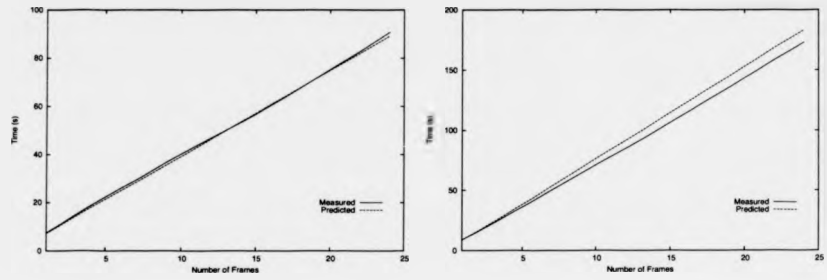
To solve this issue, two sets of profiling information from compressing both simple and complex video streams were compared. It was found that the two most data dependent areas of the algorithm were within the temporal and arithmetic encoders, as these encoders compare previous frames to the current frame. The constant data dependent values within the subtasks associated with these encoders were replaced by a parameter '`Pprevframe`' (line 26 Listing 3.6). The value of this parameter is the probability that the block being compared in the previous frame is a match with the relevant block in the current frame. This parameter was added to the list of model parameters and may be changed at the start of evaluation.

To calculate the value of this parameter, a separate application was written that searched through the input video stream and calculated the probability that all the previous frames were scanned by the temporal and arithmetic encoders. Passing this probability to the performance model prior to an evaluation resulted in accurate predictions for a number of video streams of different complexities.

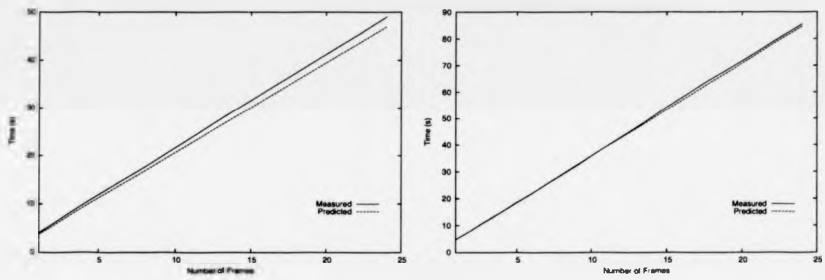
3.3 Evaluating the Performance Model

Once the algorithm's performance characterisation was complete, the model was compiled into a number of analytical models and linked into a single executable. This model was then repeatedly evaluated (on the command line) in order to achieve a set of predictive results. Within the application steering method documented later however, remote evaluation was used to enable the SMC algorithm itself to evaluate the model and predict its own performance.

Four sets of results were obtained by comparing predicted execution times using the SMC PACE performance model with the real, measured execution times obtained from compressing specific areas of DRUNKY and LIMBO video streams. Each SMC compression was achieved using the Spatial, Rice and Arithmetic encoding techniques. As the Temporal encoding technique is the most performance-critical within the SMC



Graph 3.1: A comparison between the predicted and the measured execution times of the SMC algorithm for the DRUNKY video stream: Frames 360-386, without Temporal encoding (left); Frames 850-874, with Temporal encoding (right).



Graph 3.2: A comparison between the predicted and the measured execution times of the SMC algorithm for the LIMBO video stream: Frames 37-61, without Temporal encoding (left); Frames 37-61, with Temporal encoding (right).

algorithm, two results for each video stream where Temporal encoding was and was not used are included in order to illustrate the accuracy of the model across differently-performing executions. These results are shown in Graphs 3.1 and 3.2. If the predicted execution time was the same as the real execution time for given compression parameters and video streams, the two lines on the graph would completely overlap. It was therefore preferable for the two lines in each graph to be as close to each other as possible.

Graph 3.1 shows the results of compressing different parts of the DRUNKY video stream: Frames 360-384 consists of complex camera displacements (left); Frames 850-874 consists of a less complex camera zoom (right), with different compression

parameters (with and without temporal encoding respectively).

Graph 3.2 shows the results of compressing the entire LIMBO video stream that was available (Frame 37-61). The LIMBO video stream is far simpler than the DRUNKY video stream, with fewer colours and a constant colour background for the majority of the stream. This results in quicker compressions (due to less time searching for similar elements in previous frames) and better compression ratios (due to the higher probability of finding these similar elements). Graph 3.2 compares the compression without (left) and with temporal encoding (right).

All measured execution times for the SMC algorithm were taken from an application that provides a simple interface for choosing compression parameters and presenting the compression results. Since computation is characterised at the source code level, this application had to be compiled without any compiler optimisations, as these provide inaccuracies between the final application's object code and the characterisations created by '`capp`'. If the SMC algorithm was optimised during compilation, the performance model's predicted performance would be highly inaccurate.

It was found that all sets of predictions were accurate to within 10% of the measured execution time. The performance model accurately predicted the different performance resulting from both changing the compression parameters and varying complexities within the video stream, a direct result of scanning the data prior to execution and refining data dependent areas of the model.

3.4 An SMC Application Steering Implementation

As described earlier in this chapter, application steering is a methodology whereby an application can efficiently 'steer' its own execution automatically in order to meet some environmental constraint. For example, a specific user's license for a distributed environment may limit that user to a certain amount of execution time. The efficient execution of applications within this environment would therefore be important in order

to enable this user to achieve optimum results in the available time.

Figure 3.3 describes the application steering methodology. Executing an application without implementing this methodology is shown on the left; this is where the user must both supply the correct application parameters and manually choose the resource the application will execute on, in the hope that the application executed with these parameters and on this resource will meet the time constraint. Choosing these parameters requires the user to have detailed knowledge not only of the performance of the application but of the environment as well. The right-hand diagram illustrates an application steering implementation; this is where the user simply submits the environmental constraint for the application's execution, and the parameters and resource are chosen automatically. This eliminates the need for specific detailed knowledge on the part of the user.

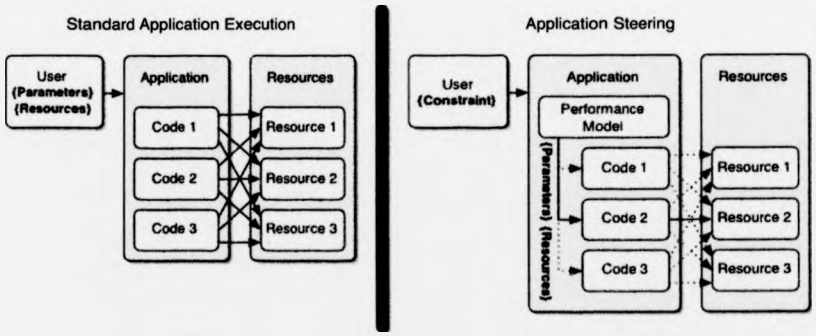


Figure 3.3: The Application Steering Methodology.

Such an application steering methodology was implemented for the SMC algorithm using the SMC performance model created in this chapter. Instead of a user manually choosing the necessary compression parameters, a single time constraint was given as input. The SMC performance model was then executed a number of times with different compression parameters, and the predicted execution time for each compression was recorded. After choosing a set of compression parameters that would provide the best compression results for the video stream within the given time constraint, the

SMC algorithm was executed. The results of implementing this methodology follow.

3.4.1 Implementation Results

Tables 3.1, 3.2 and 3.3 show the results of using application steering to achieve the optimum compression ratio in a given time constraint. Three areas of varying complexity of the DRUNKY video stream (Frames 670-689 contain a static background with a number of foreground object displacements) are used, with time constraints set between 50 and 250 seconds with 10 second intervals. Each result illustrates: the time constraint specified; the compression parameters chosen in order to meet that constraint; the predicted execution time for these chosen parameters; the measured execution time and the compression ratio obtained. Reducing both the number of previous frames and the search space, as well as turning the Arithmetic encoding off during compression, are choices made by the application steering implementation in order to keep the eventual execution time below the specified time constraint; Arithmetic encoding is switched on for the majority of higher time constraints in order to obtain the larger compression ratios possible within that time.

It is shown that the application steering implementation generally selected the appropriate compression parameters such that the SMC compression's execution stayed below the specified time constraint. The noted exception to this is where the time constraint is set too low so that even the simplest of compression parameters still resulted in an execution time that was higher that the specified constraint. As this constraint was increased, greater compression parameters were automatically chosen such that higher compression ratios could be obtained. It was found that all sets of predictions were within 28.5% (for low time constrains) and 7.1% (for high time constrains).

Time Constraint	Time Prediction	Real Time Taken	Predictive Inaccuracy	Compression Ratio	No Previous Frames	Search Space	Arithmetic Encoding
50.00	62.74	66.16	5.45%	4.67	0	1	OFF
60.00	62.74	66.34	5.74%	4.67	0	1	OFF
70.00	69.38	89.26	25.65%	4.92	0	1	ON
80.00	69.38	89.15	28.50%	4.92	0	1	ON
90.00	85.89	92.77	8.01%	5.20	1	1	OFF
100.00	92.54	111.06	20.01%	5.87	1	1	ON
110.00	92.54	110.83	19.76%	5.87	1	1	ON
120.00	92.54	111.14	20.10%	5.87	1	1	ON
130.00	92.54	111.02	19.97%	5.87	1	1	ON
140.00	92.54	110.89	19.83%	5.87	1	1	ON
150.00	92.54	111.09	20.05%	5.87	1	1	ON
160.00	92.54	110.86	19.80%	5.87	1	1	ON
170.00	92.54	110.89	19.83%	5.87	1	1	ON
180.00	92.54	111.09	20.05%	5.87	1	1	ON
190.00	182.47	181.41	0.58%	6.06	3	2	ON
200.00	182.47	181.52	0.52%	6.06	3	2	ON
210.00	182.47	181.42	0.58%	6.06	3	2	ON
220.00	210.84	201.99	4.20%	6.07	4	2	ON
230.00	210.84	202.19	4.10%	6.07	4	2	ON
240.00	237.44	220.86	6.99%	6.07	5	2	ON
250.00	237.44	220.65	7.07%	6.07	5	2	ON

Table 3.1: The time constraint predictions and compression ratios achieved from the SMC application steering implementation for the DRUNKY video stream (Frames 420-439).

Time Constraint	Time Prediction	Real Time Taken	Predictive Inaccuracy	Compression Ratio	No Previous Frames	Search Space	Arithmetic Encoding
80.00	69.92	88.49	26.56%	5.10	0	1	ON
90.00	85.12	92.96	9.21%	5.31	1	1	OFF
100.00	91.61	110.35	20.46%	6.06	1	1	ON
110.00	91.61	110.19	20.28%	6.06	1	1	ON
120.00	91.61	110.04	20.11%	6.06	1	1	ON
130.00	91.61	110.34	20.44%	6.06	1	1	ON
140.00	91.61	110.15	20.24%	6.06	1	1	ON
150.00	91.61	110.20	20.29%	6.06	1	1	ON
160.00	91.61	110.27	20.37%	6.06	1	1	ON
170.00	91.61	110.19	20.28%	6.06	1	1	ON
180.00	173.86	177.73	2.23%	6.34	3	2	ON
190.00	173.86	177.76	2.24%	6.34	3	2	ON
200.00	199.88	197.67	1.11%	6.35	4	2	ON
210.00	199.88	197.61	1.14%	6.35	4	2	ON
220.00	199.88	197.52	1.18%	6.35	4	2	ON
230.00	224.28	215.66	3.84%	6.37	5	2	ON
240.00	224.28	215.80	3.78%	6.37	5	2	ON
250.00	247.05	232.79	5.77%	6.38	6	2	ON

Table 3.2: The time constraint predictions and compression ratios achieved from the SMC application steering implementation for the DRUNKY video stream (Frames 670-689).

3.5 Summary

This chapter described the processes involved in using PACE to predict the performance of SMC, a lossless compression algorithm. First the performance of the application was characterised using CHIP^{3}S. This involved analysing the original source

Time Constraint	Time Prediction	Real Time Taken	Predictive Inaccuracy	Compression Ratio	No Previous Frames	Search Space	Arithmetic Encoding
50.00	61.27	63.39	3.46%	5.42	0	1	OFF
60.00	61.27	62.97	2.77%	5.42	0	1	OFF
70.00	67.18	82.13	22.25%	5.86	0	1	ON
80.00	67.18	82.12	22.23%	5.86	0	1	ON
90.00	88.07	102.45	16.41%	7.35	1	1	ON
100.00	88.07	102.32	16.18%	7.35	1	1	ON
110.00	88.07	102.58	16.48%	7.35	1	1	ON
120.00	88.07	102.46	16.34%	7.35	1	1	ON
130.00	88.07	102.63	16.53%	7.35	1	1	ON
140.00	88.07	102.34	16.20%	7.35	1	1	ON
150.00	144.07	156.21	8.43%	7.78	3	2	ON
160.00	144.07	156.32	8.50%	7.78	3	2	ON
170.00	162.03	171.90	6.09%	7.81	4	2	ON
180.00	178.87	186.10	4.04%	7.82	5	2	ON
190.00	178.87	186.09	4.04%	7.82	5	2	ON
200.00	194.59	199.24	2.39%	7.83	6	2	ON
210.00	209.18	211.38	1.05%	7.84	7	2	ON
220.00	209.18	211.54	1.13%	7.84	7	2	ON
230.00	222.65	222.57	0.04%	7.84	8	2	ON
240.00	238.37	246.97	3.60%	7.84	3	3	ON
250.00	238.37	247.05	3.64%	7.84	3	3	ON

Table 3.3: The time constraint predictions and compression ratios achieved from the SMC application steering implementation for the DRUNKY video stream (Frames 850-869).

code with the help of [Lopez-Hernandez03] in order to locate the performance-critical areas and characterise them individually. Inter-process communication did not need to be characterised, since SMC is a sequential algorithm. Characterisation was manually refined by recording profiling measurements during a number of executions of the algorithm, and data-dependent areas of the algorithm were identified and characterised by scanning the video stream prior to evaluation and passing a 'complexity' parameter to the model.

Once the characterisation was complete, the performance model was created and evaluated. It was shown that, using varying compression parameters and multiple video streams, these evaluations could accurately predict the performance of the SMC algorithm between 0.04% and 28.5%. Such a performance model was used within an application steering methodology to enable SMC to choose its own compression parameters and achieve optimum compression results when given an execution time constraint.

While it is possible to characterise and accurately predict the performance of distributed applications within PACE, documenting this fact was not the purpose of

this chapter, as this has been shown a number of times in other publications ([Cao99] for example). The purpose of this chapter was to introduce the reader to the detailed process of creating performance characterisations with PACE, as well as the steps required in order to create an accurate performance model. Whether the application modelled was parallel or sequential was not an issue. The research governing the extensions to PACE for Grid architectures are a direct result of the lessons learnt from characterising the SMC compression algorithm.

From the experiences detailed in this chapter, it is evident that there are areas within the PACE toolkit where characterising and predicting an application could be made less complex, take less time, be more dynamic, and mode suitable for predicting applications within Grid environments. These areas, and insights as to how they may be improved, are described in the following chapter.

Chapter 4

Proposed Improvements for a Dynamic Predictive Framework

Chapter 3 showed that it is possible to characterise and predict the performance of highly data-dependent applications using PACE with an accuracy of within 30% for a variety of input data sizes and complexities. However, achieving this level of accuracy was no simple task. From creating the SMC performance model, a number of issues with the features of PACE performance modelling were noted, and these issues used to motivate further research:

1. Prior to establishing an accurate performance model, the characterisations of data-dependent areas of the application must be continuously refined from profiling information obtained during the application's execution. While this refinement is deemed necessary, as data-dependent elements of code are generally unpredictable (unless a tool for scanning the data prior to evaluation is implemented, as documented for SMC), there is no reason why it could not be automated in some way. This would reduce the time taken to achieve the accuracy required within the model and demand less knowledge of profiling techniques on the part of the model developer.

2. During the use of '`capp`' to automate the characterisation of sequential elements

of the application, each iterative and conditional statement characterised must be associated with an expression that describes its loop count and probability of execution respectively. While data-dependent expressions can not be accurately set prior to execution, parameter-dependent expressions could be calculated in advance and set during this characterisation. Automating this calculation of parameter-dependent expressions by scanning previous assignments of variables referenced within these expressions would greatly reduce the time required to characterise sequential elements of an application and demand less knowledge of the application on the part of the model developer.

3. While sequential elements are characterised within subtasks as a control flow of machine-code instructions, these instructions are extrapolated by '`capp`' from the application's source code. This introduces inherent inaccuracies between the source code and any optimisations that may have been introduced during compilation. Describing computation from the application's compiled object code would remove these inaccuracies and result in a characterisation that would exactly match the application's eventual execution.

This chapter proposes a number of extensions to PACE in order to resolve some of these issues and make PACE more suitable for evaluating and predicting the performance of applications within dynamic, heterogeneous environments. These extensions include: a more flexible level of characterisation for the rapid creation of performance models, thus enabling the prediction of a wider set of applications; a more portable framework for performance evaluation among heterogeneous environments; the monitoring of applications in order to facilitate the automated refinement of performance characterisations; the ability to model and predict distributed Java applications, as well as describing their computation at the object code level, without the original source code; the introduction of a confidence metric that can be assigned to all predictions to depict their accuracy. The details involved in implementing these extensions for the

creation of a dynamic predictive and monitoring framework are described in Chapters 5, 6, 7 and 9.

4.1 A Flexible Characterisation Philosophy

PACE operates a strict philosophy on performance characterisation. Applications are characterised as a number of sequential computations (subtasks), each with an associated parallelisation strategy (parallel template) that describes how the sequential computation is distributed across a number of processors. An application's execution environment is characterised as a single hardware object, which limits the class of application that PACE can predict. For example, virtual-machine- and web-based applications require more in-depth performance characterisations in order to capture the performance-critical elements of the virtual machine and web server respectively; PACE currently can not characterise, and therefore predict the performance of, these classes of application.

This thesis proposes a more flexible philosophy on performance characterisation. While the layered methodology used in PACE is still adhered to, applications are characterised as a number of transactions, and the control flow of transactions within an application described by a transaction map. A transaction is defined as an item of work that is either a sequential computation, an inter-resource communication, or a combination of the two. Applications can therefore be characterised in a number of different ways from, at the least, one transaction characterising the entire application to a large number of transactions, each describing small computational elements and communications. This philosophy provides the ability to characterise a greater class of application, enabling the description of both in-depth characterisations of communication and object code (for parallel applications) and references to single transactions (for web-based applications). The method behind characterising applications is then left to the discretion of the model's developer.

In order to support this approach, the application's execution environment is characterised as an implementation of a platform interface. This implementation can access a number of platform objects that characterise performance-critical elements inherent within the class of application and its execution environment. For example, a platform interface implementation of a Hotspot JVM would be used to evaluate the performance of distributed Java applications. This implementation would access three performance objects that describe the platform's bytecode computation, inter-platform communication, and the runtime optimisations of the JVM, and the predicted performance of the platform would be obtained via the platform interface during evaluation. If required, this interface also provides an access to historical performance data. Figure 4.1 compares this new, flexible characterisation philosophy with PACE's layered framework.

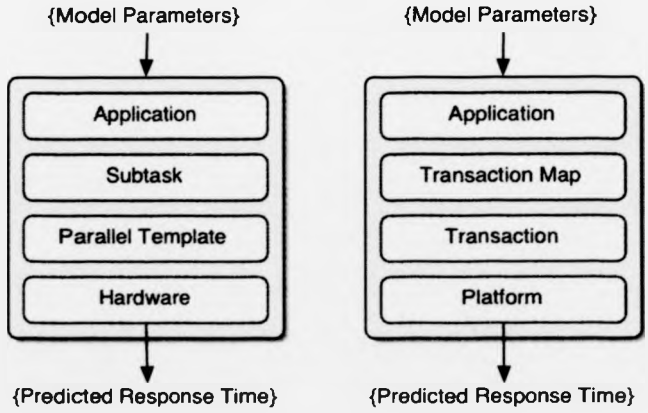


Figure 4.1: A comparison between the two layered frameworks for performance characterisation: the PACE implementation (left) and a newly developed, transaction-based philosophy (right).

Providing a more flexible approach to performance characterisation also facilitates a trade-off between the time required to develop the model and the model's eventual predictive accuracy. A transaction's characterisation can be either a single reference to an item of work that is evaluated from historical performance data (pos-

sibly less accurate), or an in-depth description of the control flow of object code that is evaluated from platform benchmark timings (assumed to be very accurate). This flexibility in performance evaluation makes this framework more suitable for a larger set of applications and performance prediction scenarios.

4.2 A Portable Predictive Framework

PACE performance characterisations are written in a non-standard language and compiled into an executable performance model. While these models are self-evaluating (that is, they evaluate on execution) and do not require any other evaluation software to calculate predictive results, they are not portable among heterogeneous environments and need to be compiled and linked for every platform where an evaluation is required. Furthermore, any necessary modifications to a model's characterisations would require a recompilation of the model.

In order to overcome this portability issue and remove the requirement for multiple performance models of the same application to be present for different architectures, it is proposed that the flexible characterisation language introduced in the previous section is implemented as an XML-based [W3C00] language, with each performance object written as an XML file. XML is the current standard for representing human-readable, text-based hierarchies and graphs of data (such as the control-flow of an application's performance-critical elements), and is supported by sophisticated parsers for schema validation and on-the-fly error checking. It is also proposed that an evaluation engine is implemented within the Java platform, enabling these XML-based performance models to be evaluated on any architecture where a JVM is installed.

Communication of XML-based performance models would be much simpler than that of the existing PACE framework. Being text-based, a collection of XML files could be efficiently compressed and easily shared among administrative domains, evaluated 'as is', and modified or instrumented as required without any necessary recompi-

lation or linking. These features of XML make it an appropriate language to represent portable and descriptive characterisations of application performance within a Grid environment. Furthermore, the Global Grid Forum are currently defining an XML-based 'standard' performance language [Gunter00a] and, in using XML, it means that this characterisation language could be easily converted to one of these new standards when they are finalised.

4.3 Automated Model Creation and Refinement

While the characterisation of an application within CHIP^{3}S is semi-automated with `'capp'`, `'capp'` still requires the developer to define the expressions for an application's loop counts and probability values that describe the control flow of computation. Furthermore, once the performance of an application has been encapsulated within a PACE performance model, this model remains static unless a developer refines the characterisation and recompiles the model. A PACE model's evaluation is always assumed to be 100% accurate, and any comparisons made between the predicted performance and the actual performance of the application during execution are the responsibility of the model's developer.

To overcome these problems, an extension to `'capp'` for the automated characterisation of transactions is proposed, as well as a framework for the automated refinement of performance models. Rather than asking the model developer for an expression each time an iterative or conditional statement is recognised, the origin of this expression is searched for within previously executed areas of the application. If an expression can be calculated from an application's previous mathematical calculations, the statement is classed as parameter-dependent and its calculated expression is inserted into the performance characterisation. If an expression cannot be calculated, it is assumed data-dependent and an initial value of 1 is used. Calculating and assigning parameter-dependent expressions during the automated characterisation of transactions

aims to reduce the time required to develop the performance model.

An extension to the Application Response Measurement (ARM) standard is proposed to monitor the performance of characterised applications in order to automatically refine performance models. Once the development of a model has been completed, transactions, and data-dependent areas of code within transactions, are instrumented with ARM API calls. These calls interface with an ARM consumer interface, which measures these performance-critical elements of the application, and populates a historical performance repository with this measured data. Periodically, this historical data is compared with the application characterisation and the platform's computational benchmark timings, and any inaccurate elements located are refined accordingly. These automated refinements aim to provide more accurate predictive results from future evaluations of the model.

This automated refinement of performance characterisations can be extremely useful for both applications that do not contain many performance-critical data-dependent elements and applications that are evaluated from historical data. For applications that are highly data-dependent, the performance data used for refinement is likely to contain large fluctuations, and will be associated with a low confidence during evaluation. Therefore, in order to predict the performance of highly data-dependent applications a method as previously described must be used where the data is scanned prior to evaluation and its nature encapsulated within the model's parameters. However, while automating such a technique is deemed possible, it is not considered part of this research.

4.4 Predicting Java Applications

While PACE characterises applications written in C, Fortran and Mathematica, the ability to characterise and predict the performance of Java applications is proposed. [Getov01] suggests the following ten reasons for using the Java platform to implement

scientific applications for Grid computing:

1. *Language.* The Java programming language includes features that are beneficial for large-scale software engineering projects, such as packages, object orientation, single inheritance, garbage collection, and unified data formats. Since threads and concurrency control mechanisms are part of the language, it is possible to express parallelism directly at the lowest user level.

2. *Class Libraries.* Java provides a wide variety of additional class libraries including functions that are essential for Grid computing, such as the capability of performing secure socket communication or message passing.

3. *Components.* A component architecture is provided through JavaBeans and Enterprise JavaBeans to enable component-based program development.

4. *Deployment.* Java's bytecode allows for easy deployment of the software through Web browsers and automatic installation facilities.

5. *Portability.* Besides the unified data format, Java's bytecode guarantees full portability following the innovative and popular 'write-once run-anywhere' concept.

6. *Maintenance.* Java contains an integrated documentation facility. Components that are written as JavaBeans can be integrated within commercially available Integrated Development Environments (IDEs).

7. *Performance.* Recent research results have proven that the performance of many Java applications can come very close to that of C or FORTRAN.

8. *Gadgets.* Java-based smart cards, Personal Digital Assistants (PDAs), and smart devices will expand the working environment for scientists.

9. *Industry.* Scientific projects are sometimes required to evaluate the longevity of technology before it can be used. Strong vendor support for Java helps make it a technology of current and future consideration.

10. *Education.* Universities all over the world are teaching Java to their students.

Not detailed here are the advantages gained in performance characterisation from the strict layout of the Java classfile format. As all Java applications must be compiled to this standard classfile format in order to conform to the JVM specification [Lindholm99], bytecode parsing and instrumentation tools can be implemented that provide the ability to analyse and modify compiled Java applications. The performance of Java applications can therefore be characterised at the object code level (thus removing any characterisation inaccuracies from compiler optimisations) and instrumented without needing the original source code (to insert transaction monitoring APIs for example).

4.5 An Associated Confidence

When evaluating a PACE performance model, there is always the assumption that the resulting predictions are within some level of accuracy. With a more flexible level of characterisation, and the ability to have very simple performance models containing limited performance descriptions in order to quickly evaluate predictions, this assumption may not always hold true. It is therefore necessary to provide the facility to determine the level of accuracy of predictive evaluations.

It is proposed that a confidence metric can be associated with predictive evaluations in order to depict their accuracy. This confidence is assigned depending on how 'trusted' both the historical data and the data-dependent elements of characterisations are during an evaluation. Whether data is 'trusted' or not is dependent upon the amount of data recorded from past executions, as well as the fluctuations of this data. Data-dependent areas of an application's characterisation whose performance

fluctuates greatly among past executions, or has not been executed frequently will be assigned a lower confidence than those areas that do not fluctuate and have been executed many times. This confidence metric can be used by performance-based middleware services in order to enable better decision-making regarding the confidence of predicted performance information.

4.6 A Dynamic Prediction and Monitoring Framework

With these extensions to PACE in mind, Figure 4.2 provides an overview of the proposed performance and monitoring framework, whose implementation details are documented in the following chapters of this thesis.

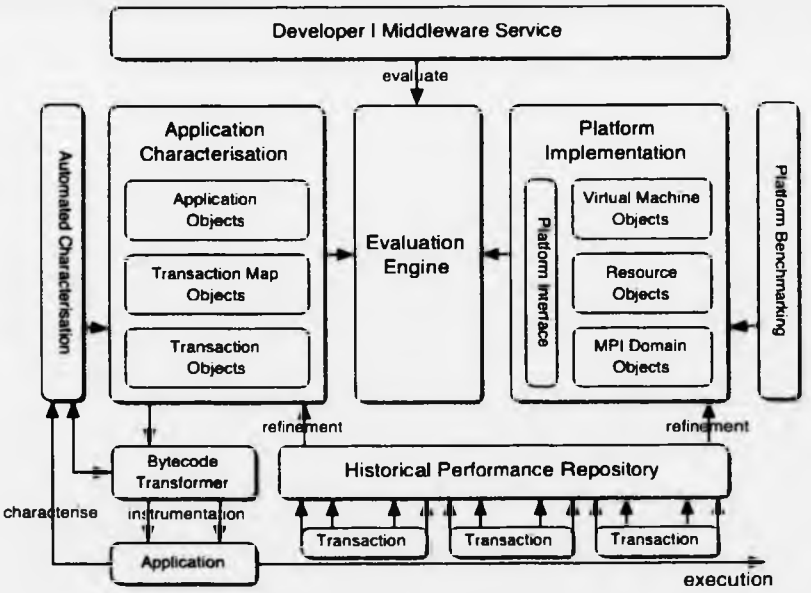


Figure 4.2: An overview of the proposed performance and monitoring framework.

In order to evaluate the prediction of an application for a given platform, the performance of both the application and the platform's execution environment must be

characterised. An application is characterised by a number of application, transaction map and transaction objects, each of which describe the control flow of performance-critical elements of the application. An automated characterisation tool can be used to easily populate a transaction object by using a bytecode transformer to capture the control flow of Java bytecode. Parameter-dependent expressions in this control flow are calculated from previous operations within the application in order to decrease the level of model developer intervention required.

A platform is characterised as an implementation of a platform interface that access a number of platform performance objects in order to evaluate the runtime performance of its execution environment. Figure 4.2 presents the implementation of a MPI-based Hotspot JVM platform that is described by resource objects (to characterise the performance of Java bytecode executing on the resource), MPI domain objects (to characterise the performance of MPI-based communication) and virtual machine objects (to characterise the runtime performance of the JVM). Each of these objects are populated by a number of platform benchmarking suites. A developer or middleware service can evaluate these performance characterisations by using the evaluation engine.

Prior to execution, any data-dependent areas characterised within transactions are located and the application is instrumented by the bytecode transformer in order to obtain information regarding their performance during execution. During execution, these calls report the required performance information to a historical performance repository, the contents of which are periodically used to refine both the transaction objects and the platform implementation's objects. Future evaluations of these characterisations inherently take these refinements into account.

4.7 Summary

This chapter described a number of proposed extensions to PACE for a performance and monitoring framework that is more suitable for the dynamic prediction and analysis of applications within Grid architectures. These extensions included: a more flexible performance characterisation language; a more portable implementation for use within heterogeneous architectures; a more sophisticated technique for the automated creation of performance models; the automated refinement of performance characterisations from historical data; the ability to predict Java applications; the possibility of assigning a confidence metric to all evaluated predictions. The details behind these extensions are documented in the following chapters of this thesis.

Chapter 5

The Performance Characterisation of Java Applications

In the previous chapter, a number of improvements were proposed in order to make PACE more suitable for predicting distributed applications within dynamic, heterogeneous environments such as those found in Grid computing. Among these suggestions were:

1. A more flexible characterisation language in order to capture and describe the performance of a broader range of applications (including both e-science and e-business applications), as well as providing the basis for decreasing the time necessary to perform such characterisations.
2. The ability to characterise and predict Java applications, an increasingly popular platform for implementing high performance applications.
3. The implementation of a platform layer that provides a standard interface to evaluate the performance of different execution environments.
4. A method of assigning a confidence to all evaluated predictions.
5. The automated refinement of performance characterisations from historical data.

The following chapters of this thesis document the implementation of these extensions as a flexible and dynamic performance prediction and monitoring framework for distributed Java applications (known as 'jPACE'). This chapter introduces the jPACE Performance Characterisation Language ('jPCL') and describes how it can be used to capture the performance-critical elements of applications. Chapters 6 and 7 describe the characterisation of execution environments and the implementation of an evaluation engine used to calculate predictive results from performance models respectively. Chapter 8 documents the use of 'jPACE' for the performance characterisation and accurate prediction of three MPI-based scientific kernels and Chapter 9 describes the automated refinement of characterisations after monitoring their performance during execution.

This chapter consists of two sections:

1. A detailed description of the jPACE Performance Characterisation Language for the performance characterisation of Java applications. Using a more flexible transaction-based approach is shown to facilitate the description of a broader class of application, as well as allowing a trade-off between the time it takes to create a performance model and its eventual predictive accuracy. Each layer of the jPCL framework is described, together with how they relate to each other to implement an application's performance model. A simple case study is used to illustrate the construction of a performance model using this characterisation language.

2. A description of a number of tools that have been implemented to further reduce the time needed to characterise the performance of an application. These tools include the parsing of a compiled application's Java bytecode in order to locate and describe, within a jPCL transaction, any computation and communication present within the application, as well as the automatic calculation of expressions to characterise parameter-dependent iterative and conditional areas

of bytecode. The automated characterisation of a number of Java methods is used as an example.

5.1 The jPACE Performance Characterisation Language

As previously documented, CHIP^{3}S employs a layered framework for performance characterisation. Each layer describes performance-critical hardware and software elements of the application: sequential computation; how these sequential elements are distributed among the available hardware; and the hardware itself. Each performance-critical element of the application is described in the CHIP^{3}S language as a performance object, and every object associated with a performance model is compiled and linked to form a single executable which, when evaluated, returns predictive results based on the model's input parameters.

With CHIP^{3}S the model developer must fulfill a number of requirements before any predictive evaluations are possible, namely that every performance-critical element of the application being modelled, including every atomic unit of computation and every communication present, must first be characterised. Forcing this amount of detail into the performance model can result in longer model creation times and may not be necessary if, for example, a large amount of trusted historical performance data is available.

The same layered methodology for performance characterisation is used in jPCL. However, transaction-based characterisation is introduced as a way of overcoming these restrictions, providing a basis for characterising a broader class of application and reducing the time needed to create performance models. While CHIP^{3}S imposes a strict separation between elements of computation and the parallelisation of these elements among the available hardware, applications are modelled in jPCL as a number of transactions or items of work. A transaction can characterise computation, communication, or a combination of the two, with the relation between transactions within

an application characterised by transaction maps. There is no set limit to a transaction's size, making it possible to characterise an application as one large transaction (requiring a less detailed understanding of the application, a smaller time in creating the model but possibly resulting in lower predictive accuracy) or as a large number of very small transactions similar to the ideology of CHIP^{3}S (requiring a more detailed understanding of the application and a larger model creation time, but resulting in greater predictive accuracy). This compromise between the time taken to create the performance model and the model's predictive accuracy is left to the model developer's discretion.

Detailed performance characterisations of computation within transactions are similar to those of the characterisation of subtasks within CHIP^{3}S. Iterative and conditional areas of bytecode are characterised with expressions that can either be related to global parameters of the performance model (if parameter-dependent) or set to a constant value (for unpredictable, data-dependent areas of code). Such detailed characterisation within transactions is however optional, further reducing the minimum time needed to create performance models. Transactions can also be defined as a single reference to a Java method within a class and evaluated according to historical information collected during the method's previous executions. This type of characterisation is most useful when describing transactions with small response times (such as those within an e-business framework) and is necessary for native methods where the detailed characterisation of Java bytecode is not possible.

This section discusses how the performance of Java applications can be described using jPCL. The performance of an application's execution environment is also achieved in jPCL within the platform layer, and includes the performance of the resource's computation, inter-resource communication and (in the case of Java) the runtime optimisations of the JVM, all described as a number of jPCL performance objects. Figure 5.1 shows the layered framework used in jPCL for performance characterisation. A description of the platform layer and its associated objects is documented

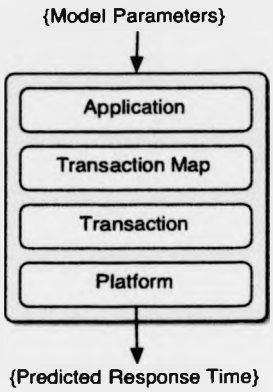


Figure 5.1: A layered methodology for application characterisation including: an application layer for defining global model parameters and the platforms that the model will be evaluated on; a transaction map layer that characterises the execution order of, and the communication between, transactions; a transaction layer for the characterisation of all transactions within an application; a platform layer that describes the performance of the application's execution environment.

in Chapter 6.

A detailed description of the three layers involved in characterising applications in jPCL follows. For each layer, a performance object is shown from an example performance model constructed from a simple sequential sorting algorithm. The original source for this algorithm is shown in Listing 5.1.

5.1.1 Application Layer

As is the case with CHIP[1]S, a performance model includes just one application object that is defined as the entry point to the model's evaluation. It is this application object that defines the model's global parameters, the platforms the application is to be evaluated on, and a control flow for evaluating transaction maps. The response time that results from evaluating the application object of a performance model is the predicted response time of the characterised application.

An application object can contain a number of the following declarations:

```
1
2   public void sort() {
3
4       for (int i = 0; i < a.length - 1; i++)
5           for (int j = a.length - 2; j >= i; j--)
6               if (a[j] > a[j+1])
7                   swap(j, (j + 1));
8
9   }
10
11  public void swap(int x, int y) {
12
13      int temp = a[x];
14      a[x] = a[y];
15      a[y] = temp;
16
17  }
18
```

Listing 5.1: A Java implementation of a simple algorithm for sorting an (initially random) integer array ('a').

1. *Platform Declaration.* Equivalent to the '`hrduse`' option in the CHIP^{3}S application object, the '`platform`' declaration defines a platform that the application will be evaluated on. A separate '`platform`' declaration is required for every platform that is used during the execution of the application. For example, a distributed application that executes on 16 resources will require 16 '`platform`' declarations within its application object in order to evaluate the performance of each of these resources; for a sequential application only one declaration is necessary. Changing the number of '`platform`' declarations defined during a predictive evaluation provides insight into the performance scalability of distributed applications. An example of a '`platform`' declaration taken from the sorting algorithm's application object is shown in Listing 5.2.

```
7
8     <jPACE:platform resource="mscs-25.dcs.warwick.ac.uk"
9                     vm="sun-hotspot-i386-linux-1.4.1_01"/>
10
```

Listing 5.2: An example '`platform`' declaration that defines the platform's resource hostname and Java virtual machine.

2. *Confidence Declaration.* Declares the maximum value that a confidence metric, assigned to all predictive response times, can be set to; the minimum value

is always 0. This value must be declared such that a maximum confidence can be assigned to trusted performance characterisations during evaluation. Each transaction defines their own confidence control flow that is used to evaluate transaction-specific confidence assignments, and these assignments must fall between 0 and this maximum value. An example '`confidence`' declaration is shown in Listing 5.3.

```
10
11    <jPACE:confidence max="1"/>
12
```

Listing 5.3: An example '`confidence`' declaration that constrains all confidence assignments to be between 0 and 1.

3. *Variable Declaration.* Declares a variable for use within the current performance object. Variables are optional within all performance objects and any number of variables may be declared. Variables are used within a performance characterisation in the same way that they are used in any other programming language. In order to control the flow of statements during a model evaluation, the variable's value can be manipulated by using '`setVariable`' statements. This allows parameter-dependent variables within an application to be characterised and evaluated appropriately (such as the height and width of a frame in the SMC example documented in Chapter 3, as well as any parameterised inter-platform communication). An example of a '`variable`' declaration taken from the sorting algorithm's transaction object is shown in Listing 5.4

```
5
6     <jPACE:variable name="NElem"/>
7
```

Listing 5.4: An example '`variable`' declaration that defines a variable called '`NElem`' whose initial value is not defined. The value of this variable can be accessed and modified anywhere within the performance object where it is declared.

A variable's initial value can be set to either a constant value or an expression that contains any other variables that have been previously declared. Within these

expressions, any modifier surrounded by the characters '${' and '}' is evaluated to the current value of the variable whose name is equal to that modifier. If the variable's initial value is not set, it must be initialised prior to being accessed during an evaluation.

4. *Parameter Declaration.* Declares a parameter to the performance model. Parameters provide the same functionality as variables, although they can only be defined within an application object and their initial value can be modified at the start of evaluation without any change to the performance model. An example parameter declaration is shown in Listing 5.5.

```
12
13    <jPACE:parameter name="NElem" value="1000"/>
14
```

Listing 5.5: An example '`parameter`' declaration that defines a parameter called '`NElem`' whose initial value (assuming it is not modified prior to evaluation) is set to 1000. A '`parameter`' declaration must define its initial value.

5. *Link Declaration.* Equivalent to the '`link`' declaration in CHIP[3]S, a '`link`' declaration allows variables declared in other performance objects to be initialised prior to their evaluation. Variables in other performance objects can therefore be appropriately initialised by the current object in order to correctly control the evaluation. An example '`link`' declaration is shown in Listing 5.6.

```
14
15    <jPACE:link targetObject="bs.tranmap" targetVariable="NElem" newValue="${NElem}"/>
16
```

Listing 5.6: An example '`link`' declaration stating that when the performance object '`bs.tranmap`' is evaluated by this performance object, a variable declared in '`bs.tranmap`' called '`NElem`' will be set to the current value of the '`NElem`' variable in this performance object.

6. *Proc Declaration.* Equivalent to the '`proc exec`' declaration in CHIP[3]S, the '`proc`' declaration provides the facility to describe the control flow of performance characterisations within a performance model. A '`proc`' can contain

statements that are used both to initialise object variables prior to the object's evaluation and to evaluate other performance objects. The majority of statements used within a '`proc`' declaration (including '`for`' and '`if`' statements for implementing control flow, examples of which are shown in Listings 5.7 and 5.8 respectively) do not result in a predicted response time, but aim to initialise performance objects prior to and during their evaluation. Only statements within a '`proc`' declaration that evaluate other performance objects result in predicted response times.

```
1
2     <jPACE:for variable="i" startValue="1" endValue="${nP} - 2" increment="${i} + 1">
3
```

Listing 5.7: An example '`for`' statement. Each statement contained within the body of the '`for`' statement is evaluated, while the value of the control variable declared (in this case '`i`') is in the range of the evaluated expressions '`startValue`' and '`endValue`' inclusive. The control variable is modified after each iteration as stated by the '`increment`' attribute.

```
3
4     <jPACE:if leftExpression="${p_array_rows}*${nP}" condition="GREATER_THAN"
5               rightExpression="${array_rows}">
6
```

Listing 5.8: An example '`if`' statement. Each statement contained within the body of the '`if`' statement is evaluated once, provided that the condition declared is true at the time of the statement's evaluation. Currently supported '`condition`' values are '`EQUALS`', '`GREATER_THAN`', '`GREATER_THAN_OR_EQUALS_TO`', '`LESS_THAN`' and '`LESS_THAN_OR_EQUALS_TO`'. '`NOT_EQUALS`' and boolean operations such as '`AND`', '`OR`' and '`NOT`' are currently not supported so far as they have not been required within any developed performance characterisations. Adding support for these operations requires a quick and simple extension to the evaluation engine's '`if`' statement object.

Each performance object must have at least one '`proc`' declaration called '`main`' that provides an entry point to the evaluation of that object. When a performance object is evaluated, it is the total evaluated response time from the control flow of its entry '`proc`' declaration that is returned as the object's evaluated response time. Therefore the response time calculated from evaluating the application object's entry '`proc`' declaration is the predicted response time for the entire performance model.

The complete application object taken from the sorting algorithm's performance model can be seen in Listing 5.9. The object contains: one '`platform`' declaration for predicting the characterised application on the resource '`mscs-25`' running Sun's Linux Hotspot JVM version 1.4.1_01; one parameter to the model called '`NElem`' that defines the number of elements in the array to be sorted and is set to an initial value of 1000; one '`link`' declaration that initialises the value of the variable '`NElem`' in the performance object '`bs.tranmap`' to the value of the variable '`NElem`' in this application object (in this case 1000, if not altered by the user or environment prior to evaluation); the '`proc`' declaration that serves as the entry point to the performance model.

```
1
2   <?xml version="1.0" encoding="UTF-8"?>
3
4   <!-- Simple bubblesort sequential characterisation -->
5
6   <jPACE:application>
7
8     <jPACE:platform resource="mscs-25.dcs.warwick.ac.uk"
9                     vm="sun-hotspot-i386-linux-1.4.1_01"/>
10
11    <jPACE:confidence max="1"/>
12
13    <jPACE:parameter name="NElem" value="1000"/>
14
15    <jPACE:link targetObject="bs.tranmap" targetVariable="NElem" newValue="${NElem}"/>
16
17    <jPACE:proc name="main">
18      <jPACE:evaluateTransactionMap transactionMap="bs.tranmap" platforms="ALL"/>
19    </jPACE:proc>
20
21  </jPACE:application>
22
```

Listing 5.9: The application performance object from the characterised sorting algorithm's performance model.

The entry point '`proc`' declaration in this application object contains only one statement (the '`evaluateTransactionMap`' statement), which is used to evaluate a transaction map object and retrieve its evaluated response time (in this case the '`bs.tranmap`' transaction map). The statement's '`platforms`' attribute is used to tell the evaluation engine which platforms (declared in the application object) the transaction map should be evaluated for. If necessary, it is possible to evaluate mul-

tiple transaction maps for different platforms, and this can be achieved by setting the '`platforms`' attribute. In this example, the transaction map is to be evaluated on all the declared platforms which, since the sorting algorithm is sequential, is the single platform defined.

5.1.2 Transaction Map Layer

Where the application object describes the control flow of transaction maps within the performance model, a transaction map object describes the control flow of transactions. However, the transaction map object also characterises any communication that may occur between transactions. Transaction map objects are evaluated from the performance model's application object.

A transaction map object can contain a number of the following declarations:

1. *Variable Declaration.* Used to declare variables for use within the transaction map object.

2. *Link Declaration.* Allows the values of variables declared in other performance objects to be initialised when that performance object is evaluated.

3. *Proc Declaration.* Allows the control flow of performance characterisation to be described. A transaction map must have at least one '`proc`' declaration called '`main`' that serves as the entry point to the evaluation of the object. The calculated response time gained from evaluating this '`proc`' is the response time of the transaction map.

4. *Map Declaration.* Equivalent to the control flow found in the parallel template object in CHIP^{3}S, a '`map`' declaration consists of a number of statements that describe the control flow of transactions and the communication between them. However, while a parallel template is associated with only one subtask, a transaction map can describe the evaluation of any number of transactions. This number

will be influenced by both the model creator's inclination and the class of application being characterised.

Similar to that of the parallel template, a '`map`' declaration is constructed as a control flow of '`step`' declarations, each of which describe a concurrently occurring element of work or communication (such as transactions executing concurrently on multiple resources or a number of threads running concurrently within the same virtual machine). It is possible to nest multiple '`step`' declarations within a '`map`' declaration, as well as defining control flow with '`for`' and '`if`' statements such that the most complex of communication strategies within an application can be characterised. An example '`map`' declaration taken from a characterised scientific kernel described later in this thesis is shown in Listing 5.10.

Inter-platform MPI communication is characterised with the use of '`MPI`' statements ('`MPISsend`' and '`MPIRecv`' in this example). Both of these statements define: their corresponding API call (the '`MPISsend`' statement defines the destination API as '`Recv`' so as to characterise an MPI communication between '`Ssend`' and '`Recv`' APIs); their source and destination platforms, defined as an expression evaluated to a platform index; the communication datatype; and the size of the communication performed. Platform indexes relate to the order in which the platforms were declared in the application object (the first and last platforms declared referenced by indexes 0 and '`${nP} - 1`' respectively) and to the platforms which were passed to this transaction map by the corresponding application object's '`evaluateTransactionMap`' statement.

A '`for`' statement is used within this map declaration in order to characterise multiple communications of the same type between the range of platforms that this transaction map is being evaluated on. The '`if`' statement is used in order to modify the size of the communication to the final platform (platform index

```
12
13  <jPACE:map name="crypt2.map">
14
15    <jPACE:step>
16      <jPACE:evaluateTransaction transaction="crypt2-init.tran" platforms="0"/>
17    </jPACE:step>
18
19    <jPACE:setVariable variable="p_array_rows"
20                       value="(((${array_rows} / 8) + ${nP} - 1) / ${nP}) * 8"/>
21    <jPACE:for variable="i" startValue="1" endValue="${nP} - 2" increment="${i} + 1">
22      <jPACE:step>
23        <jPACE:MPISsend destAPI="Recv" source="0" dest="${i}"
24                        datatype="MPI.BYTE" size="${p_array_rows}"/>
25      </jPACE:step>
26    </jPACE:for>
27    <jPACE:if leftExpression="${p_array_rows}*${nP}" condition="GREATER_THAN"
28              rightExpression="${array_rows}">
29      <jPACE:setVariable variable="p_array_rows"
30                         value="${p_array_rows} - ((${p_array_rows}*${nP}) -
31                                                   ${array_rows})"/>
32    </jPACE:if>
33    <jPACE:step>
34      <jPACE:MPISsend destAPI="Recv" source="0" dest="${nP} - 1"
35                      datatype="MPI.BYTE" size="${p_array_rows}"/>
36    </jPACE:step>
37
38    <jPACE:step>
39      <jPACE:evaluateTransaction transaction="crypt2-encrypt.tran"
40                                 platforms="1 -- (${nP} - 1)"/>
41    </jPACE:step>
42    <jPACE:step>
43      <jPACE:evaluateTransaction transaction="crypt2-encrypt.tran"
44                                 platforms="1 -- (${nP} - 1)"/>
45    </jPACE:step>
46
47    <jPACE:setVariable variable="p_array_rows"
48                       value="(((${array_rows} / 8) + ${nP} - 1) / ${nP}) * 8"/>
49    <jPACE:for variable="i" startValue="1" endValue="${nP} - 2" increment="${i} + 1">
50      <jPACE:step>
51        <jPACE:MPIRecv sourceAPI="Ssend" source="${i}" dest="0"
52                       datatype="MPI.BYTE" size="${p_array_rows}"/>
53      </jPACE:step>
54    </jPACE:for>
55    <jPACE:if leftExpression="${p_array_rows}*${nP}" condition="GREATER_THAN"
56              rightExpression="${array_rows}">
57      <jPACE:setVariable variable="p_array_rows"
58                         value="${p_array_rows} - ((${p_array_rows}*${nP})
59                                                   - ${array_rows})"/>
60    </jPACE:if>
61    <jPACE:step>
62      <jPACE:MPIRecv sourceAPI="Ssend" source="${nP} - 1" dest="0"
63                     datatype="MPI.BYTE" size="${p_array_rows}"/>
64    </jPACE:step>
65
66    <jPACE:step>
67      <jPACE:evaluateTransaction transaction="crypt2-finialise.tran"
68                                 platforms="0"/>
69    </jPACE:step>
70
71  </jPACE:map>
72
```

Listing 5.10: An example '`map`' declaration defining a number of '`step`' declarations that characterise the control flow of inter-platform MPI communication and transactions. Transactions are evaluated by the '`evaluateTransaction`' statement, which defines both the name of the transaction object to be evaluated and the platforms to evaluate the transaction on. In this example, the '`crypt2-encrypt.tran`' transaction is defined as executing concurrently on multiple platforms to capture the behaviour of a parallel MPI application.

'${nP} - 1').

The complete transaction map object taken from the sorting algorithm's performance model can be seen in Listing 5.11. The transaction map object defines: one variable called '`NElem`', whose value is passed by the '`link`' declaration in the application object; a '`link`' declaration to initialise the value of '`NElem`' passed from the application object to the variable '`NElem`' in the transaction object '`bs-kernel.tran`'; the entry '`proc`' declaration to the transaction map object that evaluates the '`map`' declaration called '`kernel`'. The '`map`' declaration contains one '`step`' declaration, which evaluates the '`bs-kernel.tran`' transaction on all the platforms passed from the application object.

```
1
2   <?xml version="1.0" encoding="UTF-8"?>
3
4   <jPACE:transactionMap>
5
6     <jPACE:variable name="NElem"/>
7
8     <jPACE:link targetObject="bs-kernel.tran" targetVariable="NElem"
9                                               newValue="${NElem}"/>
10
11    <jPACE:proc name="main">
12      <jPACE:evaluateMap map="kernel"/>
13    </jPACE:proc>
14
15    <jPACE:map name="kernel">
16
17      <jPACE:step>
18        <jPACE:evaluateTransaction transaction="bs-kernel.tran"
19                                   platforms="ALL"/>
20      </jPACE:step>
21
22    </jPACE:map>
23
24  </jPACE:transactionMap>
25
```

Listing 5.11: The transaction map performance object from the characterised sorting algorithm's performance model.

5.1.3 Transaction Layer

As previously stated, transaction objects characterise performance-critical items of work present within the modelled application. There is no limit to their size, and

transactions can characterise computation and/or inter-resource communication on any number of platforms.

A transaction object can contain a number of the following declarations:

1. *Variable Declaration.* A '`variable`' declaration is used to declare variables for use within the transaction object.

2. *Proc Declaration.* Allows the control flow of performance characterisation to be described. A transaction must have at least one '`proc`' declaration called '`main`' that serves as the entry point to the evaluation of the object. The calculated response time gained from evaluating this '`proc`' is the response time of the transaction.

3. *Confidence Declaration.* A '`confidence`' declaration contains a number of statements that define the evaluated confidence of the transaction. During the transaction's evaluation, if any of its elements are associated with a confidence, the value of this confidence is evaluated from the control flow defined within the transaction's '`confidence`' declaration. This declaration can be as simple or as complicated as required in order to characterise any possible confidence calculation. An example '`confidence`' declaration is shown in Listing 5.12.

```
1
2    <jPACE:confidence variable="c"">
3      <jPACE:setVariable variable "c" value="${nE} / 1000"/>
4    </jPACE:confidence>
5
```

Listing 5.12: An example '`confidence`' declaration that defines an evaluated confidence for all elements characterised within this transaction. This confidence evaluates to the number of executions ('`${nE}`') divided by 1000. The value of this confidence is restricted as defined within the '`confidence`' declaration in the application object.

A '`confidence`' declaration contains a '`variable`' attribute that defines a variable whose post-evaluation value will be used as the statement's confidence. In the given example, the variable '`c`' is defined, so it is the evaluated value of

'c' that is used to assign a confidence. Furthermore, an 'nE' modifier can be used here in order to reference the transaction's previous number of executions, as stated within transaction's historical data.

4. *Method Declaration.* Equivalent to the control flow found within a subtask's 'proc cflow' statement in CHIP^{3}S, a 'method' declaration consists of the performance characterisation of the control flow of a method's bytecode and/or inter-platform communication APIs. Each 'method' declaration consists of four attributes that declare the method's class, name, descriptor and type. A 'method' declaration can be of one of three types: 'characterised', where the declaration contains the complete control-flow of bytecode and communication that characterises this transaction's work; 'transaction', where the declaration is just a reference to historical data; 'native', to characterise a native method. An example 'method' declaration is shown in Listing 5.13.

```
22
23    <jPACE:method class="uk.ac.warwick.dcs.hpsg.applications.misc.bubblesort.BubbleSort"
24                  method="sort" descriptor="()V" type="characterised">
25
```

Listing 5.13: An example 'method' declaration that defines a characterised method named 'sort()V' in the class 'uk.ac.warwick.dcs.hpsg.applications.misc.bubblesort.BubbleSort'.

During the execution of a Java application, frequently executed blocks of bytecode are optimised by the Java virtual machine in order to improve the application's performance. Due to this optimisation of blocks rather than individual bytecodes, methods are characterised as a control flow of blocks of bytecode represented within a 'method' declaration by a 'bytecodeBlock' statement. Bytecode blocks are defined as sequences of bytecode within a method that do not contain any conditional branch instructions or method invocation opcodes. The reason behind characterising blocks in this manner is to enable the parameterisation of conditional and iterative elements of the method within transac-

tions; this would not be possible if these bytecode blocks included conditional bytecodes. An example '`bytecodeBlock`' statement is shown in Listing 5.14.

```
25
26      <jPACE:bytecodeBlock id="sort()V:1"/>
27
```

Listing 5.14: An example '`bytecodeBlock`' statement defining an '`id`' '`sort()V:1`' that associates the statement with the appropriate benchmark timings. Such timings constitute a resource performance object's characterisation.

The control flow of bytecode blocks within the method is characterised with the use of '`loop`', '`case`' and '`MPIcase`' statements. Iterative elements of computation (such as '`for`' and '`while`' statements) within a method are characterised by a '`loop`' statement, with an associated loop count expression to describe the number of times a section of bytecode is repeatedly executed. Conditional elements of computation (such as '`if`' and '`switch`' statements) within a method are characterised by a '`case`' statement. A '`case`' statement contains a number of '`probValue`' statements, each of which is associated with an attribute that describes the probability that a section of bytecode will execute. If a conditional statement used within an application is based on the current platform (or rank) within an MPI application, then a '`MPIcase`' statement is used instead of a '`case`' statement; '`probValue`' statements used within an '`MPIcase`' statement contain an expression to designate the platform, or range of platforms, that the '`probValue`' statements execute upon. Both the loop count and probability attributes for the '`loop`' and '`case`' '`probValue`' statements can be either constant (data-dependent) or an expression related to the values of any variables declared within the transaction (parameter-dependent areas of code). These statements can be specified as data-dependent with an optional '`data_dependent`' attribute, the value of which is used both during the evaluation of models to assign a confidence metric and the automated refinement of characterisations.

Communication characterised within transactions is declared by the same '`MPI`' statements previously shown in the transaction map object except that, to simplify the characterisation of transactions, the source and destination platform index are not specified. Any inter-platform communication is evaluated as the average communication of that type and size between the current platform and all the other platforms that the current transaction is being evaluated upon. For an MPI-based parallel application running on a homogeneous cluster where the communication performance between all platforms is similar, this method of evaluation remains accurate. For more complex control flows of communication, such as between heterogeneous clusters, it is more appropriate to characterise the communication within a transaction map.

The compiled bytecode of the '`sort`' method shown in Listing 5.1 and its associated performance characterisation, taken from the performance model's transaction object, is shown in Figure 5.2. A more detailed explanation of Java bytecode is found later in this chapter. The '`bytecodeBlock`' statements characterise the blocks of sequential bytecode that implement this method. Other methods that are invoked from within a characterised method are evaluated by the '`callMethod`' statement. The loop counts associated with the two '`loop`' statements (that characterise the two '`for`' loops in the application's original source) are an expression of the '`NElem`' variable. This variable is declared as a parameter to the model and defines the size of the array being sorted ('`a.length`' in the original source code). The probability value associated with the '`probValue`' statement that characterises the '`if`' statement in the original source is set to '`0.5`' and is flagged as data-dependent, as the data within the array to be sorted is initially random. A tool that has been implemented to automate the performance characterisation of '`method`' declarations from compiled bytecode, as well as being a first step in calculating the parameter-dependent expressions for iterative and conditional statements, is documented later in this chapter.

It should be noted that unconditional branch opcodes (such as '`goto`') *are* in-

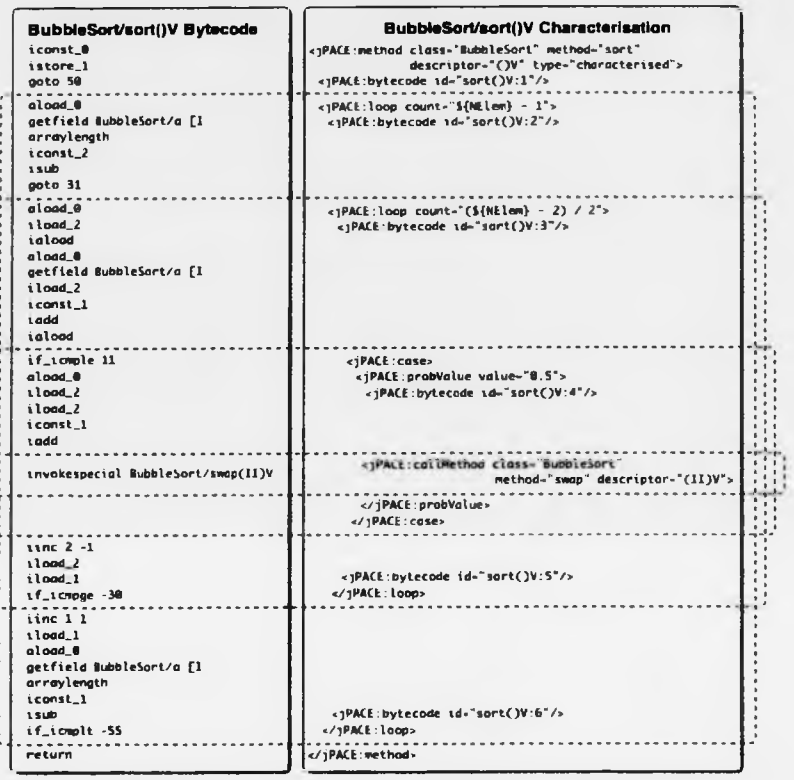


Figure 5.2: The compiled Java bytecode of the '`sort`' method from the sorting algorithm (left) and its associated performance characterisation as a '`method`' declaration within a transaction object (right).

cluded in bytecode blocks. The bytecode executed after the branch is also included within the same block until a conditional branch or method invocation opcode is found. Therefore, the bytecode block '`sort()V:1`' from Figure 5.2 is defined as the first three opcodes of the method, as well as the opcodes starting from '`iload_1`' (the opcode jumped to by the '`goto`' opcode, also part of bytecode block '`sort()V:6`') until the condition branch opcode at the end of the method ('`if_icmplt`').

The transaction object taken from the sorting algorithm's performance model is shown in Listing 5.15. The transaction object declares one variable called '`NElem`' that

is used in characterising the '`loop`' statements within the '`method`' declaration, and an entry '`proc`' declaration that acts as the entry point to the object's evaluation and contains an '`evaluateMethod`' statement that evaluates the transaction's '`sort`' method. A '`confidence`' declaration is included to associate all evaluations within this transaction with the value 1. The characterised '`swap`' method, evaluated by the '`callMethod`' statement in the '`sort`' method, is also shown.

5.2 The Automated Characterisation of Transactions

Due to the fact that all performance-critical elements of work present within an application are characterised as transactions in jPCL, it is often the case when developing a performance model that characterising these transactions represents the majority of the model creation time. When endeavouring to reduce the time needed to create performance models, it is therefore important that there are tools available to help automate the characterisation of transactions.

This section describes the two main techniques implemented as a tool in this research for automating the characterisation of '`method`' declarations within transaction performance objects. This tool is the jPACE equivalent of '`capp`', which translates C source code into '`clc`' instructions used within a subtask's '`proc cflow`' declarations. However, while '`capp`' does significantly reduce the time needed to develop a model's subtask when using PACE, there are two significant problems in using '`capp`' for automating performance characterisation:

1. '`capp`' extrapolates the equivalent machine code's '`clc`' instructions from the application's source code. This creates '`proc cflow`' declarations that do not incorporate any optimisations made by the compiler; in fact it is recommended within the PACE documentation that applications being modelled should be compiled with all compiler optimisations turned off. This is likely to be a major drawback when applying these techniques in the high performance community.

```
1
2  <?xml version="1.0" encoding="UTF-8"?>
3
4  <jPACE:transaction>
5
6    <jPACE:variable name="NElem"/>
7
8    <jPACE:confidence variable="c">
9      <jPACE:setVariable variable="c" value="1"/>
10   </jPACE:confidence>
11
12   <jPACE:proc name="main">
13     <jPACE:evaluateMethod
14                 class="uk.ac.warwick.dcs.hpsg.applications.misc.bubblesort.BubbleSort"
15                 method="sort" descriptor="()V"/>
16   </jPACE:proc>
17
18   <jPACE:method class="uk.ac.warwick.dcs.hpsg.applications.misc.bubblesort.BubbleSort"
19                 method="swap" descriptor="(II)V" type="characterised">
20     <jPACE:bytecodeBlock id="swap(II)V:1"/>
21   </jPACE:method>
22
23   <jPACE:method class="uk.ac.warwick.dcs.hpsg.applications.misc.bubblesort.BubbleSort"
24                 method="sort" descriptor="()V" type="characterised">
25
26     <jPACE:bytecodeBlock id="sort()V:1"/>
27
28     <jPACE:loop count="${NElem} - 1">
29       <jPACE:bytecodeBlock id="sort()V:2"/>
30
31       <jPACE:loop count="(${NElem} - 2) / 2">
32         <jPACE:bytecodeBlock id="sort()V:3"/>
33
34         <jPACE:case>
35           <jPACE:probValue value="0.5" data_dependant="yes">
36
37             <jPACE:bytecodeBlock id="sort()V:4"/>
38             <jPACE:callMethod
39                 class="uk.ac.warwick.dcs.hpsg.applications.misc.bubblesort.BubbleSort"
40                 method="swap" descriptor="(II)V"/>
41
42           </jPACE:probValue>
43         </jPACE:case>
44
45         <jPACE:bytecodeBlock id="sort()V:5"/>
46       </jPACE:loop>
47
48       <jPACE:bytecodeBlock id="sort()V:6"/>
49     </jPACE:loop>
50
51   </jPACE:method>
52
53 </jPACE:transaction>
54
```

Listing 5.15: The transaction performance object from the characterised sorting algorithm's performance model.

2. When '`capp`' characterises iterative and conditional statements from the original application, the statement's loop counts and probability values must be specified either by inserting '`pragma`' statements into the application's source code or by entering their values at the command line during '`capp`'s' execution. As

documented in Chapter 3, calculating these values can be time-consuming and generally requires an in-depth knowledge of the application concerned. There is currently no method used by '`capp`' to automatically calculate and set these loop counts and probability values during performance characterisation.

The jPACE equivalent of '`capp`' aims to overcome these issues. Firstly, due to the unique nature of compiled Java applications as a number of class files that must conform to the Java Virtual Machine specification, it is possible with the use of a Java class file parser to characterise and instrument an application's methods at the bytecode level. Characterising at this level means that any optimisations made by the compiler are inherently taken into account.

Secondly, when any iterative or conditional statements are found during this bytecode characterisation, an attempt is made to determine their loop counts and probability values relative to any parameters in the application. The area of bytecode associated with these statements that determines whether to perform another iteration of their execution (or even execute their bytecode in the first place) is located and calculated relative to previous assignments of the variables concerned. If, for example, the loop count of an iterative statement within a method is determined to be solely related to a specific calculation that occurs previously within the execution of the application, this statement's loop count is assumed to be parameter-dependent. This calculation is then entered as the statement's loop count. If this loop count cannot be calculated, it is assumed to be data-dependent and is set to a pre-defined constant value. Once the automated transaction characterisation has been completed, data-dependent values can either be set to an appropriate value by the model developer, or updated automatically by the automated refinement of performance models documented in Chapter 9.

Automating the characterisation of parameter-dependent statements when characterising transactions can greatly reduce the time needed to develop a performance model. In some cases it can also remove the necessity to execute the application being modelled prior to its prediction, in order to obtain the relevant profiling information

required to complete the model (as shown in Chapter 3). The details behind the implementation of this automated characterisation of Java bytecode are documented below.

5.2.1 An Introduction to Java Bytecode

A compiled Java class file has a specific structure that describes a class's fields and methods, a number of associated attributes for optional descriptive data and a constant pool that serves as a lookup table for all elements within the class [Lindholm99]. As most elements of a class (individual bytecodes, method declarations and so on) are defined by reference strings which are often referred to more than once within a class, each element refers to an index within the constant pool in order to simplify the class and reduce its size. A Java class file parser (similar in design to CFParse [IBM03a] and the Byte Code Engineering Library (BCEL) [Apache03]) has been implemented in Java as part of this research in order to read these elements and the data encoded within a Java class and store them in Java objects for either reading or modifying. It is possible to parse a class, modify the code embedded within these Java objects in some pre-determined manner, and rewrite this instrumented class such that its behaviour can be changed without any recompilation of the original source. It is this parser that is used to read a method's Java bytecode during the automated characterisation of jPACE transactions, as well as the instrumentation of bytecode for monitoring and application profiling purposes documented in Chapter 9.

Methods are defined within '`Method_Info`' elements and can contain a number of attributes including: a '`LineNumberTable`' attribute for referencing a method's bytecode to a line number from the original source code; a '`LocalVariableTable`' attribute that references variables used within the method's bytecode to the actual names of the associated variables in the original source code; a '`Code`' attribute that contains the actual compiled bytecode of the method itself. The '`LineNumberTable`' and '`LocalVariableTable`' attributes only exist if debugging is set during compilation. A method's bytecode currently consists of 206 pre-defined opcodes that

represent any calculation or class interaction possible within the bounds of the Java Language Specification [Gosling00]. The Java Virtual Machine emulates a stack-based processor, with references to class instantiations, arrays and datatypes stored in a method's local variables and any calculation or interaction between these local variables implemented via stack operations. All opcodes used for mathematical operations, branch instructions and method invocations perform their function on references to objects currently on the stack. An example of this is shown in Listing 5.16, which shows the bytecode used in adding two integers found in a method's local variables 0 and 1 respectively and storing the result in local variable 2, all via the stack.

```
1
2   0 iload_0     ; push the integer in local variable 0 onto the stack
3   1 iload_1     ; push the integer in local variable 1 onto the stack
4   2 iadd        ; pop two integers off the stack, add them together,
5                 ; and push the result onto the stack
6   3 istore_2    ; pop off an integer from the stack and store it in local variable 3
7
```

Listing 5.16: An example stack-based operation for adding two integers.

Iterative and conditional areas of bytecode are implemented using the '`goto`' opcode and a number of branch opcodes that compare either one or two values currently on the stack. From looking at a method's bytecode, it is generally clear which areas of bytecode were compiled from '`for`' loops, '`while`' loops, '`if`' statements and so on, mainly due to a number of common compiling techniques used among the majority of Java compilers.

For example, the Java bytecode for a compiled '`for`' loop statement of the order '`for (int i = 0; i < 5; i++) {...}`' is shown in Listing 5.17. The statement's initialisation expression '`int i = 0`' is compiled to the opcodes on lines 2 and 3, where 0 is stored in local variable 2, the local variable chosen by the compiler to hold the variable named '`i`'. The '`goto`' opcode then jumps over the body of the '`for`' loop in order to check whether the statement meets its condition before it is executed. The '`if_icmplt`' opcode at line 13 then branches back to the beginning of the

body (the instruction following the '`goto`' opcode) of the '`for`' loop while the variable '`i`' is less than 5 (5 is pushed onto the stack prior to the branch instruction). The increment expression '`i++`' is compiled to the opcode '`iinc 2 1`', which increases the value of local variable 2 ('`i`') by one prior to each condition of the loop. Other iterative statements are compiled in similar ways to the example shown. '`while`' loops are compiled as above except that there is no initialisation code, and '`do/while`' loops do not have the initial '`goto`' opcode since the body of the statement is always executed once prior to the conditional expression.

```
1
2   7       iconst_0       ; push 0 onto the stack
3   8       istore_2       ; pop an integer off of the stack and store it in local
4                          ; variable 2
5   9       goto 14        ; unconditionaly jump a branch offset of 14 (23)
6   12
7
8   ** FOR LOOP BODY **
9
10  20      iinc 2 1       ; increment the integer in local variable 2 by 1
11  23      iload_2        ; push the integer in local variable 2 onto the stack
12  24      iconst_5       ; push 5 onto the stack
13  25      if_icmplt -13  ; branch an offset of -13 (12) if local variable 2 is
14                         ; less than 5
15
```

Listing 5.17: An example of the Java bytecode for a compiled '`for`' loop statement.

Compiled conditional statements such as the '`if/else`' statement are distinguishable from iterative statements as they contain positively branching conditional opcodes to jump over the code if the condition fails to be met. Listing 5.18 shows an example of the Java bytecode from a compiled statement of the nature '`if (i == 0) { ... } else { ... }`', where the variable '`i` is held in local variable 1. The branch instruction '`ifne`' at line 3 jumps over the body of the '`if`' statement if the variable '`i`' is not equal to zero; otherwise (if the variable '`i`' is equal to zero and thus the instruction '`ifne`' does not branch) there is a '`goto`' opcode that jumps over the body of the '`else`' statement.

There are four method invocation opcodes. Prior to executing any of these opcodes, the arguments of the method that is about to be executed are pushed onto

```
1
2   2       iload_1  ; load the integer in local variable 1
3   3       ifne 9   ; branch an offset of 9 (12) if the value of
4                    ; local variable 1 is not equal to 0
5   6
6
7   ** BODY OF IF **
8
9   9       goto 6   ; unconditionaly branch an offset of 6 (15)
10  12
11
12  ** BODY OF ELSE
13
14  15
15
```

Listing 5.18: An example of the Java bytecode for a compiled '`if`' statement.

the stack in the order that they are defined within the method's declaration. When the method invocation opcode is executed, these arguments are popped off the stack and placed in the new method's local variables in this same order for access within the invoked method. Once the called method has finished executing normally (ie. without an exception or error being thrown), any value that it may return is popped onto the calling method's stack.

5.2.2 Bytecode Characterisation

There are a number of tools [Gupta00, vanVliet03] currently available that enable the decompilation of un-signed class and jar files into Java source code very similar to the source of the original application. These tools work by parsing the Java class file and scanning the class's methods for known techniques (as shown above) used when compiling Java source code.

In automating the characterisation of '`method`' declarations within jPCL transactions, a similar method to these decompilers is used, with jPCL performance characterisations resulting instead of Java source code. A jPCL transaction is given as input to the Automated Characterisation Tool (ACT) and any empty '`method`' declarations of type '`characterised`' that are found within the transaction are characterised appropriately. A number of XML parsers (similar in structure to the Java class file

parser) have been implemented in order to read and manipulate the jPCL performance objects used within a performance model. All object statements and declarations are parsed and the appropriate Java objects are populated with their information and nested statements. As with the Java class file parser, these objects can then be modified appropriately and rewritten back to an XML file in order to automate the instrumenting of jPCL performance objects.

On characterising a transaction, an attempt is made to locate the class defined by the '`method`' declaration. The jPACE environment includes a configuration file that is used to define classpaths, performance object paths, the amount of debugging output required and so on, that are used by the automated characterisation tool and evaluation engine. It is within these defined classpaths that an attempt is made to locate the class. It is possible to define these locations as URLs if a remote path is required. An example jPACE configuration file is shown in Listing 5.19.

Assuming the declared class is found, it is parsed and the '`Method_Info`' object that describes the method whose name and descriptor are stated within the transaction's '`method`' declaration is located. The method's bytecode is then extracted from the method's '`Code`' attribute. This bytecode is then characterised as shown in Figure 5.3 (Figure 5.2 shown previously is an example of an automatically characterised transaction from the Bubblesort '`sort()V`' method) and the jPCL '`method`' declaration is populated as appropriate. At any one time during characterisation a section of the method's bytecode is characterising an individual jPCL statement or declaration. For example: when characterising the entire method's bytecode as a section, the characterised statements are added to the '`method`' declaration; when characterising a section of bytecode that repeats iteratively during execution, the characterised statements are added to the current '`loop`' statement; and so on. A method is populated by stepping through each bytecode of the current section and adding each opcode to the current bytecode block. When a specific opcode that denotes an iterative or conditional area of bytecode is found, a new statement appropriate to this opcode is added to

```
1
2  <?xml version="1.0" encoding="UTF-8"?>
3
4  <!-- jPACE configuration file -->
5  <jPACE:configuration>
6
7  <!-- jPACE debug output setting -->
8  <jPACE:debug output="MAX"/>
9  <!--  <jPACE:debug output="STANDARD"/> -->
10 <!--  <jPACE:debug output="MIN"/> -->
11 <!-- <JPACE:debug output="NONE"/> -->
12
13 <!-- paths to any performance model objects -->
14   <jPACE:modelPath dir="jPACE/perfmodels/dhpc/ep"/>
15   <jPACE:modelPath dir="jPACE/perfmodels/dhpc/fft"/>
16   <jPACE:modelPath dir="jPACE/perfmodels/jgf/crypt1"/>
17   <jPACE:modelPath dir="jPACE/perfmodels/jgf/crypt2"/>
18   <jPACE:modelPath dir="jPACE/perfmodels/jgf/sparsematmult"/>
19   <jPACE:modelPath dir="jPACE/perfmodels/misc/bubblesort"/>
20
21 <!-- paths to any platform definition objects -->
22   <jPACE:platformPath dir="jPACE/platforms"/>
23
24 <!-- paths to any bytecodeBlock benchmark timings -->
25   <jPACE:benchmarkPath dir="jPACE/benchmarks"/>
26
27 <!-- paths to any bytecodeBlock definition objects -->
28   <jPACE:bytecodeBlocksPath dir="jPACE/bytecodeblocks"/>
29
30 <!-- paths to any classes for automated characterisation -->
31   <jPACE:classPath dir="lib/classes"/>
32   <jPACE:classPath dir="lib/jdk-classes"/>
33
34 <!-- wildcard definition of classes that should not be characterised -->
35 <!-- any methods will be defined as of type transaction -->
36   <jPACE:noCharacteriseClasses class="java.*"/>
37
38 </jPACE:configuration>
39
```

Listing 5.19: An example jPACE configuration file.

the current statement or declaration being characterised. The section of bytecode that is defined by this new statement is then recursively characterised as a new section of bytecode.

Such an explanation demands an example. If a conditional branch opcode with a negative branch offset is found while stepping through the current section of bytecode, this always denotes an iterative area of code (a negative branch offset suggests that if the condition is met then some previous bytecode already executed will be executed again). An iterative area of bytecode is characterised within the jPCL as a '`loop`' statement, and therefore a '`loop`' statement is initialised and added to the current statement or declaration being characterised. The section of bytecode that

Figure 5.3: The method used by the ACT in characterising a method's bytecode.

is characterised by this new '`loop`' statement will start from the opcode where the branch opcode's negative branch offset points to and ends at the branch opcode itself. This new, smaller section of the bytecode is then extracted from the previous section. The ACT then steps through this new section of bytecode from the beginning, adding any characterised statements found to this new '`loop`' statement. A similar process is performed for conditional areas of bytecode where the branch offset is positive (characterised as '`case`' statements) and for invoke opcodes (characterised as '`callMethod`' statements), where the method being invoked is found within its designated class and characterised in the same manner. Once the characterisation of this new method has finished, the '`method`' declaration for the called method is added to the transaction.

If a '`method`' declaration is of type '`transaction`' or '`native`', it is not characterised and, if it contains any statements, they are removed. It is also possible to mark certain methods, or all the methods within certain classes, as type '`transaction`' so that while a '`method`' declaration is still added to the transaction, it does not get characterised and is marked as type '`transaction`' within its declaration. This is achieved by setting a number of '`noCharacteriseClasses`' elements within the jPACE configuration file. As can be seen on line 36 of Listing 5.19, all classes that start with the phrase '`java.`' will be declared of type '`transaction`'. If any native methods are called by the method currently being characterised, these are automatically declared of type '`native`' and are not characterised either. Any invoke opcodes to MPI methods are characterised as the appropriate '`MPI`' statement.

When these specific opcodes are found, a unique ID is given to the current bytecode block and a '`bytecodeBlock`' statement is added to the current statement or declaration being characterised. The bytecodes stored within this block and their arguments are then stored in an XML file of bytecode blocks for the method's class and given the same ID assigned to the '`bytecodeBlock`' statement. It is this file that is used to benchmark all the bytecode blocks that exist within a performance model for

a given platform prior to its evaluation. Part of such a bytecode blocks file is shown in Listing 5.20; both the opcode's local variable arguments and whether the method is static or not are stored within this file in order to create a context within the bytecode block's benchmark similar to that of the original application. Documentation regarding the benchmarking of bytecode blocks can be found later in this chapter.

```
25
26    <jPACE:bytecodeBlock id="test([D[D[I[I[DI[I[D)V:4" staticMethod="yes">
27
28      <jPACE:OPCODE_aload localVariable="7"/><jPACE:OPCODE_aload_2/>
29      <jPACE:OPCODE_iload localVariable="10"/><jPACE:OPCODE_iaload/>
30      <jPACE:OPCODE_dup2/><jPACE:OPCODE_daload/>
31      <jPACE:OPCODE_aload localVariable="4"/><jPACE:OPCODE_aload_3/>
32      <jPACE:OPCODE_iload localVariable="10"/><jPACE:OPCODE_iaload/>
33      <jPACE:OPCODE_daload/><jPACE:OPCODE_aload_1/>
34      <jPACE:OPCODE_iload localVariable="10"/><jPACE:OPCODE_daload/>
35      <jPACE:OPCODE_dmul/><jPACE:OPCODE_dadd/>
36      <jPACE:OPCODE_dastore/><jPACE:OPCODE_iinc localVariable="10" increment="1"/>
37      <jPACE:OPCODE_iload localVariable="10"/><jPACE:OPCODE_iload localVariable="8"/>
38      <jPACE:OPCODE_if_icmplt/>
39
40    </jPACE:bytecodeBlock>
41
```

Listing 5.20: An example bytecode block written during automated characterisation.

5.2.3 Parameter-Dependent Variable Calculation

Once a transaction has been characterised, the ACT tries to assign parametric expressions to every loop count and probability value found within characterised '`loop`' and '`probValue`' statements respectively. This is achieved by looking at the conditional expression associated with these statements and backtracking through the execution of the application until a complete expression can be constructed. Any conditional expressions that cannot be expressed relative to the declared variables using this technique are assumed to be data-dependent and are set to a pre-defined constant value, to be refined either by the model developer or automatically after the application's first execution.

As it currently stands, there are a number of limitations to automatically distinguishing and calculating parameter-dependent expressions within '`method`' declara-

tions. Loops where more than one control variable used within the conditional expression of the statement is modified during the loop's body are currently not supported and are defined as data-dependent. With loops where only one variable is modified during each iteration, if this loop is modified by a combination of both multiplication/division '*and*' addition/subtraction operations, or by any other non-standard mathematical operation (such as '`Math.pow`' for example), this is also not supported and is defined as data-dependent. Any '`probValue`' condition that is found to have a probability of any value other than 1 or 0 is assumed to be data-dependent, as calculating average probabilities for expressions over the course of a program is difficult, time-consuming and sometimes inaccurate. If a probability is found to be related to the current rank of platform (defined by the MPI command '`MPI.COMM_WORLD.Rank()`') however, the '`case`' statement is changed to an '`MPIcase`' statement, with all the '`probValue`' statements' '`value`' attributes changed to a '`platform`' attributes. Despite these limitations, this technique does still automate the parameterisation of a large number of '`loop`' statements and helps to reduce the time taken to characterise transactions: 79% of all '`loop`' statements were characterised automatically by this tool for the five JavaGrande benchmarks documented in Chapters 8 and 9.

A further restriction of this tool is that searching for the complete expression is restricted to the methods declared within the transaction. In order to search '`method`' declarations, this tool keeps a call tree of all the methods that called this method in order to relate expressions from calculations made in previous methods. If an expression is related to arguments passed to the top-level '`method`' declaration within a transaction, and if not all of the local variable names within the current expression match the transaction's '`variable`' declarations, the expression is also assumed to be data-dependent.

Calculating the loop count of a '`loop`' statement is achieved by first locating the two values used within the loop's conditional expression and then attempting to calculate their values prior to this iterative selection of bytecode. If these two values

have been pushed onto the stack from local variables, all previous opcodes within the method prior to this iterative selection of bytecode that access these local variables are studied in order to formulate an expression. For example, if one of the local variables was last populated with the result of the stack operation '1 + 2', it is known that the current value of this local variable is 3. However, if the last populated result is a stack operation that adds the values of two other local variables, the method is searched for opcodes that access these local variables, and so on. This continues until either a constant, or an expression that contains nothing but local variables whose names (looked up in the method's '`LocalVariableTable`') match variables declared within the transaction, is found for the local variables used within this loop's conditional expression.

When these expressions are calculated, the body of the loop is searched for any other opcodes that affect the value of the local variables used within the conditional expression. Assuming that only one of these variables is affected during the loop's body (the loop's control variable), the expression found above for the control variable's value prior to this iterative statement is the initial value of this control variable, and an expression is formulated for how this variable is modified during each iteration. An expression is then calculated for the other variable used in the conditional expression where it is known that the conditional expression will definitely fail and the loop stop iterating.

The loop count is then calculated according to these three expressions: the expression that calculates the initial value of the control variable prior to the loop statement ('`a`'), the expression that calculates the value of the control variable where the condition will fail ('`b`'), and the expression that dictates by how much the control variable is modified during each iteration ('`c`'). If the control variable is modified during each iteration by only addition and/or subtraction operators, '`c`' is calculated as the size of increment each iteration (ie. 2, if 2 is being added to the control variable each

iteration) and the loop count is calculated as:

$$\frac{b-a}{c} \tag{5.1}$$

whereas if the control variable is modified each iteration by only multiplication and/or division operators, '`c`' is calculated as the size of multiplication each iteration (ie. 2, if the control variable is being multiplied by 2 each iteration) and the loop count is calculated as:

$$log_c(\frac{b}{a}) \tag{5.2}$$

Two examples are used to clarify this automated parameterisation technique. Listing 5.21 shows the source code of a simple '`for`' loop whose loop count is dependent upon the values of the variables '`i`' and '`j`'. Listing 5.22 shows the compiled bytecode for this method. This '`for`' loop has an integer control variable named '`l`', whose value is set to 1 in the initialisation expression, whose value must stay below the value of '`k`', as stated in the condition expression, and whose value is incremented by 1 each iteration, as stated in the increment expression. No variables are declared within the transaction that this method is being characterised for.

```
3
4       public void example() {
5
6           int i = 2;
7           int j = 5;
8
9           int k = (i * 6) % j;
10
11          for (int l = 1; l < k; l++)
12              System.out.println("print");
13
14      }
15
```

Listing 5.21: The Java source for method '`example1()V`'.

The conditional expression for this iterative statement (lines 21, 22 and 23) branches back to the beginning of the loop if the value of local variable 4 (LV_4) is less than the value of local variable 3 (LV_3) as defined by the '`if_icmplt`' opcode. In

```
1
2  ACC_PUBLIC (16) "example" (10) "()V"
3    ** Attribute Entry 1 (11) "Code"
4       0       iconst_2
5       1       istore_1
6       2       iconst_5
7       3       istore_2
8       4       iload_1
9       5       bipush 6
10      7       imul
11      8       iload_2
12      9       irem
13      10      istore_3
14      11      iconst_1
15      12      istore 4
16      14      goto 14
17      17      getstatic java/lang/System/out Ljava/io/PrintStream;
18      20      ldc print
19      22      invokevirtual java/io/PrintStream/println(Ljava/lang/String;)V
20      22      iinc 4 1
21      28      iload 4
22      30      iload_3
23      31      if_icmplt -14
24      34      return
25
26    ** Attribute Entry 2 (13) "LocalVariableTable"
27      Start PC = 0  | Name = (14) "this" | descriptor = (15) "LParameterExample;"
28      Start PC = 2  | Name = (17) "i"    | descriptor = (18) "I"
29      Start PC = 4  | Name = (19) "j"    | descriptor = (18) "I"
30      Start PC = 11 | Name = (20) "k"    | descriptor = (18) "I"
31      Start PC = 14 | Name = (21) "l"    | descriptor = (18) "I"
32
```

Listing 5.22: The Java bytecode for method '`example1()V`'.

other words, the conditional expression for this '`for`' loop is:

$$LV_4 < LV_3 \tag{5.3}$$

An attempt is now made to calculate an expression for these local variables prior to the iterative statement. In this example, local variable 4 is set on lines 14 and 15 to a constant 1. Among local variables 4 and 3, only variable 4 is modified during the loop's body and is thus declared as the loop's control variable. The initial value of the control variable prior to the first iteration of the loop is 1 and, by definition, '`a`' from the equations above is also set to 1. The control variable is also incremented by 1 each iteration of the loop, so the value of '`c`' is also set to 1, and equation 5.1 will be used to calculate the loop count.

Variable '`b`' from this equation denotes the value of the control variable over the course of the iterations that will fail the loop's condition. In this case, this is when

local variable 4 is equal to local variable 3, and so '`b`' is set to local variable 3. Local variable 3 is set on line 13 and is the result of the remainder (line 12) of local variable 2 (line 11) and the result of the multiplication (line 10) of the constant 6 (line 9) and local variable 1 (line 8). In other words:

$$LV_3 = (LV_1 * 6)\%LV_2 \tag{5.4}$$

as seen in the original source. Lines 4, 5, 6 and 7 of the bytecode show that local variable 1 is set to 2 and local variable 2 is set to 5, so the value of local variable 3 and '`b`' is 2. Hence, the loop count of this '`for`' loop (LC) is:

$$LC = \frac{b-a}{c} = \frac{2-1}{1} = 1 \tag{5.5}$$

and the loop count of the characterised '`loop`' statement corresponding to this '`for`' loop is set to 1.

Listings 5.23 and 5.24 show the source and bytecode respectively of a slightly more complex example. The '`for`' loop in question is in the '`example2b(II)V`' method at line 29, the loop count of which is also related to variables '`i`' and '`j`'. However, these variables are arguments to the method and so it is necessary to look up the method's call tree is order to find their values. If it is seen during the course of the transaction being characterised that the method is called more than once with different arguments, the loop count is calculated for each argument (if possible) and an average of these expressions is used as the '`loop`' statement's loop count. If this is not possible then the loop count is assumed to be data-dependent. In this example, two variables ('`x`' and '`y`') are declared within the transaction.

The same process as above is used again to calculate the loop count. The conditional expression for this loop is located in the bytecode (lines 63, 64 and 65), again involving local variables 3 and 4. However, this time the condition is when local variable 4 is less than or equal to local variable 3, due to the '`if_icmple`' opcode used in the condition, ie.

$$LV_4 <= LV_3 \tag{5.6}$$

```
15
16      public static int x = 6;
17      public static int y = 11;
18
19      public void example2a() {
20
21          example2b(x, y);
22
23      }
24
25      public void example2b(int i, int j) {
26
27          int k = (i * 25) % j;
28
29          for (int l = 1; l <= k; l *= 2)
30              System.out.println("print");
31
32      }
33
```

Listing 5.23: The Java source for methods '`example2a()V`' and '`example2b(II)V`'.

Local variable 4 is again the control variable and is initialised to 1 prior to the start of the loop (lines 53 and 54), although this time it is multiplied by 2 each iteration (lines 59, 60, 61 and 62). As the control variable is being multiplied each iteration, equation 5.2 is used to calculate the loop count with variable '`a`' (the initial value of the control variable prior to the loop) set to 1. Variable '`c`' (the scale of multiplication each iteration) is in this case 2, as the control variable is multiplied by 2 each iteration.

Variable '`b`' is defined as the value where the conditional expression will fail, which in this case is the value of local variable 3 + 1. In this example, the value of local variable 3 prior to the loop (set at line 52) is the remainder (line 51) of local variable 2 (line 50) and the multiplication (line 49) of local variable 1 (line 47) and the constant value 25 (line 48), as follows:

$$LV_3 = (LV_1 * 25)\%LV_2 \tag{5.7}$$

Since this equation is the current expression of local variable 3 at the beginning of the method, and since the names of local variable 1 ('`i`') and local variable 2 ('`j`') from the method's '`LocalVariableTable`' attribute are not declared as transaction variables (and therefore an expression consisting of these variables cannot be assigned to the loop count), this expression would appear to be data-dependent. However, the cur-

```
33
34  ACC_PUBLIC (22) "example2a" (10) "()V"
35    ** Attribute Entry 1 (11) "Code"
36       0        aload_0
37       1        getstatic ParameterExample/x I
38       4        getstatic ParameterExample/y I
39       7        invokevirtual ParameterExample/example2b(II)V
40       10       return
41
42    ** Attribute Entry 2 (13) "LocalVariableTable"
43       Start PC = 0 | Name = (14) "this" | descriptor = (15) "LParameterExample;"
44
45  ACC_PUBLIC (23) "example2b" (24) "(II)V"
46    ** Attribute Entry 1 (11) "Code"
47       0        iload_1
48       1        bipush 25
49       3        imul
50       4        iload_2
51       5        irem
52       6        istore_3
53       7        iconst_1
54       8        istore 4
55       10       goto 17
56       13       getstatic java/lang/System/out Ljava/io/PrintStream;
57       16       ldc print
58       18       invokevirtual java/io/PrintStream/println(Ljava/lang/String;)V
59       21       iload 4
60       23       iconst_2
61       24       imul
62       25       istore 4
63       27       iload 4
64       29       iload_3
65       30       if_icmple -17
66       33       return
67
68    ** Attribute Entry 2 (13) "LocalVariableTable"
69       Start PC = 0  | Name = (14) "this" | descriptor = (15) "LParameterExample;"
70       Start PC = 0  | Name = (17) "i"    | descriptor = (18) "I"
71       Start PC = 0  | Name = (19) "j"    | descriptor = (18) "I"
72       Start PC = 7  | Name = (20) "k"    | descriptor = (18) "I"
73       Start PC = 10 | Name = (21) "l"    | descriptor = (18) "I"
74
```

Listing 5.24: The Java bytecode for methods '`example2a()V`' and '`example2b(II)V`'.

rent method ('`example2b(II)V`') contains two arguments, meaning that the values of these arguments are stored in local variable 1 and local variable 2 respectively (for a virtual method), and that these arguments would have been pushed onto the stack in order prior to calling the '`invokevirtual`' opcode. From the application's call tree that is constructed during the transaction method's characterisation, it is seen that this method is invoked from method '`example2a()`' (line 39). Extracting the bytecode from this method and examining the correct '`invokevirtual`' opcode that calls '`example2b(II)V`' shows that the two values used as the method's two arguments are the static variables '`x`' and '`y`' respectively from this class, which have the same

names as the variables declared within the transaction. The value of 'b' where the condition will fail is therefore:

$$b = LV_3 + 1 = ((x * 25)\%y) + 1 \tag{5.8}$$

as the variables 'x' and 'y' point to the local variables 1 and 2 respectively, and with 'a' = 1, 'b' = '((${x} * 25) % ${y}) + 1' and 'c' = 2. The loop count is therefore set to the following expression:

$$LC = log_c(\frac{b}{a}) = log_2(((\${x} * 25)\%\${y}) + 1) \tag{5.9}$$

Therefore, in this example, if the transaction variables 'x' and 'y' are set to 6 and 11 respectively, the loop count is evaluated to be 3.

This tool is aimed as a first step into automating all parameter-dependent expressions within an application. While it is currently limited to a number of situations, it nevertheless helps to parameterise a large set of commonly used loop structures. It is currently thought that this tool could be extended to provide the functionality to completely automate the parameterising of all parameter-dependent conditional and loop structures within the application; such a tool would be even more useful in helping to reduce the time taken to characterise transactions.

5.3 Summary

This chapter introduced the jPACE Performance Characterisation Language, an extension to CHIP^{3}S that is used within the jPACE environment to capture the performance of distributed Java applications. These extensions include two key features that aim to make jPCL an appropriate language for the performance characterisation and prediction of applications within a Grid environment:

1. *Flexibility.* An application is characterised as a control-flow of transactions. Transactions can describe computation, inter-platform communication or a combination of the two and, since their size is not limited, it is possible to capture

an entire application either as one transaction or as many small transactions. In addition, transactions can either contain detailed characterisations of the control-flow of performance-critical elements or be a single reference to an item of work that is evaluated from historical data. This flexibility makes it possible to characterise and predict the performance of a broader class of application that may exist within a Grid environment, as well as providing a trade-off between the model's creation time and its eventual predictive accuracy, should a quick but less accurate performance evaluation of an application be required.

2. *Portability.* The jPCL was implemented as an XML-based language and Java was chosen as the target platform for performance characterisation and prediction. Since all tools implemented within the jPACE environment for predicting the performance of Java applications are themselves written in Java, this results in portable performance models of portable applications that can be evaluated and executed upon any heterogeneous platform containing a Java virtual machine.

A tool implemented to automate the characterisation of jPCL transactions within a performance model was also documented. It was explained how the computation of compiled Java applications can be characterised such that compiler optimisations are inherently taken into account, as well as how parameter-dependent expressions within transactions can be automatically obtained. These tools aim to further reduce the time needed to implement a performance model.

In order to evaluate an application's performance characterisation on a given platform, the application's underlying execution environment must first be characterised. Details of this jPCL characterisation of platforms is documented in the following chapter.

Chapter 6

A Java Hotspot Platform Implementation

The previous chapter focused on the performance description of applications within the jPACE Performance Characterisation Language (jPCL). However, in order to predict the performance of an application upon a given platform, the performance-critical elements of this platform must also be characterised.

Chapter 4 proposed an extension to the PACE hardware model, the implementation of which is discussed in this chapter. The jPCL includes a platform layer, where a number of performance objects that characterise the performance-critical elements of the platform's execution environment reside. An execution environment's characterisation is an implementation of a platform interface that models the performance-influencing characteristics of that platform during the application's execution. Rather than the evaluation engine accessing the performance objects within the platform layer directly, a platform interface is used so that these characteristics are taken into account during a predictive evaluation. There are currently five interface calls, also documented in this chapter, that are used to obtain the evaluated response times for bytecode blocks, a transaction's historical data, and both point-to-point and collective MPI communications.

This chapter documents a Hotspot Java Virtual Machine platform implementation. This implementation includes three types of performance objects: a resource object, which contains bytecode block benchmark timings and historical data for transaction executions; an MPI domain object, which contains MPI benchmark timings for a number of point-to-point and collective inter-platform communications; a virtual machine object, which characterises the JVM's runtime optimisation strategies. This platform implementation can be used to predict the performance of any sequential or MPI-based characterised Java application running on a Hotspot JVM.

This chapter consists of three sections:

1. A description of the three types of platform objects that are currently used within the Java Hotspot platform implementation: the resource, MPI domain and virtual machine performance objects.

2. A description of the Java Hotspot JVM platform interface implementation. How this implementation models and predicts a Java application's execution, including the runtime optimisations used within modern JVMs to improve performance, is documented, as well as the five currently defined platform interface calls.

3. The details involved in populating the resource and MPI domain performance objects with accurate benchmark timings. A tool that has been implemented for the automated benchmarking of an application's bytecode blocks and the method used to benchmark inter-platform communication performance are described.

6.1 Platform Performance Objects

The platform interface implementation that characterises the runtime-performance of a Hotspot JVM currently contains three types of platform layer performance object:

a resource object, an MPI Domain object and a virtual machine object. When a platform interface call is made by the evaluation engine to this platform implementation, these performance objects are accessed in order to return an evaluated response time. The platform's performance-critical elements characterised by these objects within a Hotspot JVM execution environment are also documented in this section.

While there are currently three performance objects defined within the jPCL platform layer to characterise an execution environment, it is intended that more platform objects will be created in the future to allow the characterisation of more complex execution environments. In order to evaluate a web-based application, for example, it would be necessary to implement a platform object to characterise the performance of the web-server, and to describe communication APIs other than MPI. These performance objects would then be evaluated, via an extended version of the platform interface, by another implementation of this interface that would model these performance-influencing elements of the characterised application. This potential for extending the platform layer, in addition to the class of application that could be characterised and predicted, is the subject of future work.

6.1.1 The Virtual Machine Object

An example virtual machine performance object is shown in Listing 6.1. It states, among other things, the virtual machine's manufacturer, make and version, and characterises a number of performance-critical features that are implemented within the virtual machine in order to improve performance.

```
3
4   <jPACE:virtualMachine company="sun" arch="i386" os="linux" version="1.4.1_01">
5
6     <jPACE:runtimeOptimisation compiler="hotspot" optimiseIteration="1500"/>
7
8   </jPACE:virtualMachine>
9
```

Listing 6.1: The performance object for Sun's Linux Hotspot virtual machine, version 1.4.1_01.

The performance-critical feature currently taken into account during a predictive evaluation is the 'adaptive compilation' [Sun02a] of bytecode during execution. Modern JVMs determine the blocks of bytecodes that are frequently executed (called 'hotspots') and compile these to native code in order to dramatically improve the performance of future bytecode block executions. One feature of this adaptive compilation algorithm is the optimisation of a method once it has been executed a specific number of times [Suganuma00]. This number of iterations is used as a runtime optimisation metric to characterise this adaptive compilation algorithm for a given virtual machine so that it can be taken into account during predictive evaluations. For example, the '`runtimeOptimisation`' element in Listing 6.1 states that this virtual machine uses the '`hotspot`' adaptive compilation algorithm for runtime optimisation, and that bytecode blocks for this virtual machine are compiled to native code after 1500 iterations. How this metric is obtained is documented later in this section.

In Chapter 8, it is shown that a good degree of accuracy can be achieved by just characterising the adaptive compilation performance features that occur during the execution of a Java application. However, other performance-critical features of a JVM exist, including garbage collection, heap allocation and monitor contention. Modelling these features is the subject of future work. If these elements were described within a virtual machine object, it would follow that the predictive accuracy obtained from evaluations would improve.

6.1.2 The Resource Object

Resource performance objects contain a number of '`classTiming`' declarations that characterise the performance of Java computation for that resource. Each declaration consists of a number of '`methodTiming`' declarations. These in turn consist of a number of '`bytecodeBlockTiming`' declarations that describe the response time of bytecode blocks found within methods over the course of the application. The response time of a method is measured during execution if the '`monitorExecution`'

attribute is set, and is characterised by the '`noExecutions`' and '`avResponseTime`' '`methodTiming`' attributes. These values are used for method declarations of type '`transaction`' or '`native`' where a detailed performance characterisation of the method does not exist, as well as evaluating a transaction's confidence. Part of a resource object for the resource '`mscs-25.dcs.warwick.ac.uk`', that is defined as being a member of the '`mscs-dcs.warwick.ac.uk`' MPI domain, is shown in Listing 6.2.

```
1
2  <?xml version="1.0" encoding="UTF-8"?>
3
4  <jPACE:resource name="mscs-25.dcs.warwick.ac.uk" mpiDomain="mscs.dcs.warwick.ac.uk">
5
6    <jPACE:classTiming
7              class="uk.ac.warwick.dcs.hpsg.applications.misc.bubblesort.BubbleSort">
8
9      <jPACE:methodTiming method="sort" descriptor="()V"
10                         noExecutions="34684" avResponseTime="345021.75"
11                         monitorExecution="yes">
12
13       <jPACE:bytecodeBlockTiming id="sort()V:1">
14         <jPACE:iterationTiming noIterations="1" executionTime="343.1379"/>
15         <jPACE:iterationTiming noIterations="1500" executionTime="343.1379"/>
16         <jPACE:iterationTiming noIterations="1501" executionTime="5.3761"/>
17         <jPACE:iterationTiming noIterations="6000" executionTime="5.3761"/>
18       </jPACE:bytecodeBlockTiming>
19
```

Listing 6.2: A portion of an example resource performance object, characterising the performance of bytecode block '`sort()V:1`' from the '`sort`' method. All timings are in nanoseconds.

Each '`bytecodeBlockTiming`' declaration contains a number of '`iterationTiming`' declarations that characterise the variance of the block's response time, depending on the number of times it has been executed. It is possible to declare many '`iterationTiming`' declarations here in order to capture varying levels of optimisation that may occur to a bytecode block during the application's execution. Each '`bytecodeBlockTiming`' currently contains four '`iterationTiming`' declarations that describe the response time of the block before and after optimisation. However, this could be updated in the future to include a more detailed characterisation of JVM runtime optimisation. In the resource object shown in Listing 6.2, the response time of the bytecode block '`sort()V:1`' before and after optimisation (set at 1500 it-

erations) is 343.1379 and 5.3761 nanoseconds respectively. This resource's associated virtual machine object states that the runtime optimisation of bytecode blocks takes place after 1500 iterations, as declared in Listing 6.1.

6.1.3 The MPI Domain Object

MPI domain performance objects contain a number of '`MPIToDomainTiming`' declarations that characterise MPI communication between this and another MPI domain. Each '`MPIToDomainTiming`' in turn contains a number of '`MPITiming`' declarations that describe the performance of either a set point-to-point or a collective communication between the declared MPI domains. An example of an '`MPIDomain`' performance object is shown in Listing 6.3. The attributes '`intBytes`' and '`doubleBytes`' define the size of an integer and of a double respectively for that specific platform as a number of bytes. This is necessary when characterising a platform other than Java where these attributes may vary between resources; these values remain constant throughout all Java Virtual Machines for all platforms, as stated in the JVM specification. In this example, for this domain, an MPI communication of one integer is equivalent to that of four bytes.

In the same way that a '`bytecodeBlockTiming`' declaration contains a number of '`iterationTiming`' declarations that characterises the range of performance of bytecode blocks, an '`MPITiming`' declaration contains a number of '`commTiming`' declarations that characterise the range of communication performance. The more '`commTiming`' declarations defined, the more accurate the evaluated response time of a given size of communication. In this example, sending 512 bytes via a '`Ssend`' and '`Recv`' to a resource of the same MPI domain will be evaluated as taking 244812.8 nanoseconds.

It is important here to contain a number of '`commTiming`' declarations for both latency- and bandwidth-bound communication in order to accurately evaluate small and very large communications. The response times stated within these declara-

```
1
2  <?xml version="1.0" encoding="UTF-8"?>
3
4  <jPACE:MPIDomain name="mscs.dcs.warwick.ac.uk">
5
6    <jPACE:MPIToDomainTiming domain="mscs.dcs.warwick.ac.uk"
7                             intBytes="4" doubleBytes="8">
8
9      <jPACE:MPITiming sourceAPI="Ssend" destAPI="Recv" datatype="MPI.BYTE">
10       <jPACE:commTiming size="1" responseTime="194316.8"/>
11       <jPACE:commTiming size="2" responseTime="168691.2"/>
12       <jPACE:commTiming size="4" responseTime="168614.4"/>
13       <jPACE:commTiming size="8" responseTime="162521.6"/>
14       <jPACE:commTiming size="16" responseTime="169932.8"/>
15       <jPACE:commTiming size="32" responseTime="166028.8"/>
16       <jPACE:commTiming size="64" responseTime="175500.8"/>
17       <jPACE:commTiming size="128" responseTime="185830.4"/>
18       <jPACE:commTiming size="256" responseTime="199936.0"/>
19       <jPACE:commTiming size="512" responseTime="244812.8"/>
20       <jPACE:commTiming size="1024" responseTime="313587.2"/>
21       <jPACE:commTiming size="2048" responseTime="430784.0"/>
22       <jPACE:commTiming size="4096" responseTime="611609.6"/>
23       <jPACE:commTiming size="8192" responseTime="941798.4"/>
24       <jPACE:commTiming size="16384" responseTime="1630220.8"/>
25       <jPACE:commTiming size="32768" responseTime="3075801.6"/>
26       <jPACE:commTiming size="65536" responseTime="6003161.6"/>
27       <jPACE:commTiming size="131072" responseTime="1.19773184E7"/>
28       <jPACE:commTiming size="262144" responseTime="2.3797312E7"/>
29       <jPACE:commTiming size="524288" responseTime="4.7536E7"/>
30       <jPACE:commTiming size="1048576" responseTime="9.47396864E7"/>
31     </jPACE:MPITiming>
32
```

Listing 6.3: A portion of an example MPI domain performance object, characterising the performance of a point-to-point communication (MPI APIs '`Ssend`' to '`Recv`') for a varying number of byte array sizes. All timings are in nanoseconds.

tions aim to take into account the network performance of the MPI domain specifically where the benchmarking took place. Inter-domain communication, where network traffic can traverse several networks with different bandwidth and latency performance, is more difficult to characterise and predict and is the subject of future work.

6.2 The Platform Interface & Its Implementation

An evaluation engine that evaluates the predicted performance of applications characterised within jPCL and executing upon a defined platform is documented in the following chapter. As previously stated, a platform interface that the evaluation engine accesses in order to obtain a predicted performance of specific aspects of the platform's execution environment has been defined. This interface currently defines five method

calls, each of which is shown in Listing 6.4.

```
1
2   package uk.ac.warwick.dcs.hpsg.pace.perfmodel;
3
4   import uk.ac.warwick.dcs.hpsg.pace.EvaluationException;
5
6   public interface PlatformInterface {
7
8       public double getBytecodeBlockResponseTime
9           (String cls, String method, String descriptor, String id,
10           double loopCount, double probValue)
11          throws EvaluationException;
12
13      public double getMethodAvResponseTime
14          (String cls, String method, String descriptor)
15          throws EvaluationException;
16
17      public double getMethodNoExecutions
18          (String cls, String method, String descriptor)
19          throws EvaluationException;
20
21      public double getMPIDomainPToPResponseTime
22          (String sourceAPI, String destAPI, Platform destPlatform,
23           String datatype, double size)
24          throws EvaluationException;
25
26      public double getMPIDomainCollectiveResponseTime
27          (String collectiveAPI, String datatype, String function,
28           String noPlatforms, double size)
29          throws EvaluationException;
30
31  }
32
```

Listing 6.4: The platform interface, which defines five method calls used by the evaluation engine to evaluate a platform's bytecode block, transaction, and MPI communication performance. Each method returns an evaluated predicted response time of the specified performance element in nanoseconds.

This section describes an implementation of this platform interface that characterises the execution environment of MPI-based Java applications. While most of the interface calls simply provide access to the benchmark timings within the platform implementation's performance objects, the '`getBytecodeBlockResponseTime()`' interface call is implemented in order to model variances in performance that occur during the execution of a Java application due to the adaptive compilation of bytecode. The implementation returns just an evaluated response time; a confidence associated with these response times is calculated by the evaluation engine. Implementation-specific documentation of each interface call follows.

6.2.1 `getBytecodeBlockResponseTime()`

The '`getBytecodeBlockResponseTime()`' platform interface method is used to evaluate the response time of a method's bytecode blocks. Passed to this method is the class, name and descriptor of the method the bytecode block originated from, as well as the bytecode block's ID. This information is used to obtain the four '`iteration Timing`' declarations that characterise the pre- and post-optimisation response times of this block from the platform's resource object. Which of these declarations is returned as the block's evaluated response time is dependent upon how many times this block has already been executed during the course of the application's evaluation.

Loop count and probability variables are also passed to this interface call. The loop count variable is passed in order to help decrease the time taken to evaluate characterised applications. As documented in Chapter 7, a method's '`loop`' statement's response time is calculated by evaluating each statement within the '`loop`' once and multiplying their total response time by the statement's evaluated loop count. In order to achieve this without invoking a platform interface call more than once, the loop count is passed to the call, and the implementation must incorporate this loop count when calculating the bytecode block's response time. The probability variable is the current probability of this block executing within the application, as defined by a '`probValue`' statement. Both the loop count and probability value are made available to the implementation so that a record of the number of times a specific block of bytecode has been executed can be kept. The implementation is also required to incorporate this probability variable when calculating the evaluated response time.

Since one call to this interface can represent many iterations and a conditional probability, the bytecode block's evaluated response time must be an equation containing both of these loop count and probability arguments. An evaluated response time for a given iteration of a bytecode block is equal to the block's evaluated response time multiplied by both the loop count (so that multiple iterations of the block are taken into account) and the probability value (ensuring that the probability of this iteration

executing in the first place is taken into account). For example, a bytecode block with a loop count of 4 and a probability of executing equal to 0.5 will be evaluated as twice the bytecode block's response time since, although that part of the program will execute 4 times, on average the block will execute on only two of those iterations.

In order to predict the adaptive compilation strategies of the virtual machine, this JVM implementation keeps a running iteration count of all the bytecode blocks that have been previously evaluated. For a given evaluation of a bytecode block, its associated iteration count is incremented by the product of the loop count and probability value, as is the case with the evaluated response time. Using the same example, a bytecode block with a loop count of 4 and a probability of executing equal to 0.5 will have its iteration count incremented by 2 since, on average, the block will be executed twice. This method also ensures that the evaluated response time during the application is proportional to the evaluated size of the application.

In order to evaluate the block's response time, both the original iteration and the number of iterations after this evaluation are compared with the runtime optimisation iteration declared within the platform's virtual machine object. The number of iterations for this platform interface call before and after optimisation is then calculated. The call's pre-optimisation evaluation time is the product of the number of iterations before optimisation, the block's pre-optimisation response time and the current probability value, while the optimised evaluation time is the product of the number of optimised iterations, the block's optimised response time and the current probability value. The evaluated response time for this platform interface method call is the sum of these pre- and post-optimisation calculations.

6.2.2 `getMethodAvResponseTime()`

The '`getMethodAvResponseTime()`' platform interface method is used to retrieve a method's previous average response time. This information is held in the '`methodTiming`' declaration within the platform's resource object and is kept up-to-

date by the monitoring framework documented in Chapter 9. Passed to this interface call is the method's class, name and descriptor, which are used to locate the correct '`methodTiming`' declaration. The value stored within the declaration's '`avResponse Time`' attribute is returned as the evaluated response time.

6.2.3 `getMethodNoExecutions()`

The '`getMethodNoExecutions()`' platform interface method is used to retrieve the number of times a method has been executed previously. Like the method's average response time, this information is stored within the '`methodTiming`' declaration and is updated by the monitoring framework. The method is located within the platform's resource and the value stored within the declaration's '`noExecutions`' attribute is returned as the evaluated response time.

6.2.4 `getMPIDomainPToPResponseTime()`

The '`getMPIDomainPToPResponseTime()`' platform interface method is used to evaluate the response time of a point-to-point MPI communication. Passed to this interface call is the source and destination APIs that constitute this communication, the destination platform, the communication's datatype (byte, integer, double, and so on) and an expression that defines the size of the communication. Using these arguments, the '`MPITiming`' declaration within this platform's MPI domain that characterises a point-to-point communication between these source and destination APIs, and between the current and destination platform's MPI domains, is found.

Within the MPI domain file, only the point-to-point communication of bytes is characterised, and communications of other primitives are evaluated from this byte datatype '`MPITiming`' declaration. The '`MPIToDomainTiming`' declaration that contains these '`MPITiming`' declarations also states, for each datatype, the size of a single primitive relative to the size of a byte; for a given platform an integer may

be declared as four bytes long, a double as eight bytes long, and so on. Using these declarations, the size of the communication for a given primitive is evaluated as the equivalent size of this communication in bytes.

Once the size of the communication in bytes has been obtained, the response time of the communication is calculated from the '`commTiming`' declarations defined within the '`MPITiming`'. The two '`commTiming`' declarations whose defined size is closest to the communication size required are extracted from the '`MPITiming`'. These declarations are either one above or one below the required size or, if the size is larger than the largest '`commTiming`' size declared, the two largest sizes are extracted. During this calculation, it is assumed that there is a straight line between the response times of these '`commTiming`' declarations. For larger communication sizes this holds true as the communication is bandwidth dominated. The evaluated response time for this communication is therefore calculated as (see Figure 6.1):

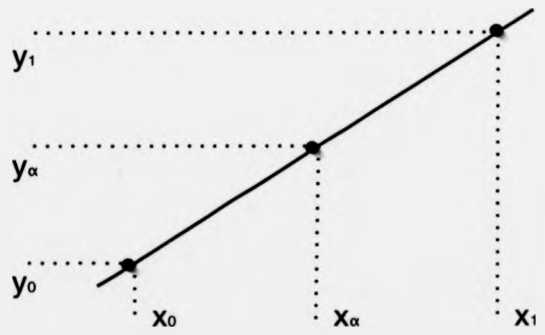


Figure 6.1: A straight line approximation between two '`commTiming`' declarations used to extrapolate the evaluated communication response time y_α.

$$y_\alpha = \frac{((y_1 - y_0) * (x_\alpha - x_0))}{x_1 - x_0} + y_1 \tag{6.1}$$

where x_0 and x_1 are the sizes of the two '`commTiming`' declarations below and above the size required respectively, y_0 and y_1 are the response times of these declarations

below and above the size required respectively, and x_α and y_α are the actual size of communication required and its calculated response time respectively. y_α is returned as the evaluated response time of this communication.

6.2.5 `getMPIDomainCollectiveResponseTime()`

The '`getMPIDomainCollectiveResponseTime()`' platform interface method is used to evaluate the response time of a collective MPI communication[1]. Passed to this interface call is the name of the collective API to evaluate, the communication's datatype and function, the number of platforms that the collective API is operating upon, and the size of the application. Using this information, the '`MPITiming`' declaration that characterises this collective communication is found within the platform's MPI domain. The evaluated response time of this communication is calculated from the '`commTiming`' declarations by, as with the point-to-point communication, assuming a straight line graph between the two closest '`commTiming`'s. This calculated response time is returned as the evaluated response time of this communication.

6.3 Benchmarking Resource and MPI Domain Objects

During an evaluation of a characterised application, it is the timings declared within the platform's performance objects that are used by the platform interface's implementation to calculate predictions. It follows that the accuracy of these timings directly affects the accuracy of the performance predictions achieved. This section documents a tool that automates the benchmarking of bytecode blocks in order to obtain a response time before and after its optimisation during execution (if optimised at all), as well as a standard method for benchmarking MPI communication for a given API call over a range of data sizes.

[1] the '`MPI_Allreduce`' collective operation is the only operation used by the five applications modelled within this thesis and is therefore the only operation that has been currently characterised

6.3.1 Benchmarking Bytecode Blocks

In order to populate a resource's '`bytecodeBlockTiming`' statement with accurate '`iterationTiming`' statements, a bytecode block benchmarking tool has been implemented. This tool takes an XML file of bytecode blocks created during the automated characterisation of bytecode as input, benchmarks each block found within the file and writes the result to the resource file associated with the platform that the benchmark is being executed upon. Predicting an application prior to its execution upon a number of different platforms requires this tool to be executed for all of these platforms so that the platform's resource objects are populated with the required bytecode block timings; after execution, historical data can be used as an alternative method of predictive evaluation.

As previously documented, it is important to capture the runtime optimisations present within the JVM if accurate performance predictions of Java applications are to be achieved. The main runtime optimisation found within modern JVMs is adaptive compilation, where frequently executed bytecode within an application is optimised dynamically during the application's execution in order to improve performance. This technique is based on a common feature of applications where 20% of the code is executed 80% of the time [Meloan99], and it therefore spends most of its time locating and optimising this 20% in order to achieve the greatest possible performance. The performance of a specific bytecode block is therefore very much related to its current context within the application.

One of the main criteria used to decide whether an application's method should be optimised is how many times the method has been executed so far during the course of the application. Each method is given an associated method count that is incremented each time it is executed. When this method count crosses a certain threshold, it signifies to the virtual machine that this method should be optimised and compiled to native code. However, if a method contains a loop, an attempt is made to calculate its loop count and if this loop count is over a certain threshold, the method is optimised

immediately before execution. This stops the inevitable case of failing to optimise a method that is only executed once but contains a loop that iterates for the majority of the application's eventual execution time [Suganuma00].

With this in mind, it was decided that the number of method executions before optimisation (assuming the method does not consist of a loop of some kind) would be declared as the metric used in characterising the performance-critical adaptive compilation algorithms of the JVM. Bytecode blocks would be iterated enough times to see where this optimisation is taking place, and it would be this number of iterations that would be declared by the '`runtimeOptimisation`' setting within the virtual machine object. This metric would also be used by the evaluation engine to predict runtime optimisations.

In order to automatically benchmark a specific bytecode block executing on a given resource, a 'place-holder' class was created; Jasmin [Meyer97], a Java bytecode assembler, was used in order to have finer control over the construction of the benchmark, part of which is shown in Listing 6.5. The class consists of a '`static void main`' method which repeatedly invokes a '`bench`' method, consisting of two calls to a native library used to store the current timestamp in nanoseconds, and printing the difference between these timestamps to the standard out. Prior to benchmarking, this benchmark class is parsed by the bytecode parser previously documented, instrumented with both the bytecode block and the required initialisation bytecode in order to pass verification, and written out to another file. This updated benchmark class is then executed, the output is captured, processed, and the corresponding '`bytecodeBlockTiming`' declaration populated accordingly within the current platform's resource object.

While instrumenting the benchmark class, the class is modified such that the bytecode block is in as similar a context as possible to that of the original application:

1. Both the '`bench`' method's descriptor and the descriptor of the '`invokevirtual`' opcode method that invokes this method are changed to the same descriptor as

```
26
27  .method public bench()V
28          .limit stack 20
29          .limit locals 50
30  ; initialise the bytecode block here
31
32  ; local variables 44,46,48 are reserved for timings
33  ; start of the benchmark
34          invokestatic NanoTimer/getClockTime()D
35          dstore 44
36
37  ; insert bytecode block here
38
39  ; end of the benchmark
40          invokestatic NanoTimer/getClockTime()D
41          dstore 46
42
43  ; substract the end time from the beginning
44          dload 46
45          dload 44
46          dsub
47          dstore 48
48
49  ; print out the value to standard out - this is caught by the benchmarking program
50          getstatic java/lang/System/out Ljava/io/PrintStream;
51          new java/lang/StringBuffer
52          dup
53          invokespecial java/lang/StringBuffer/<init>()V
54          dload 48
55          invokevirtual java/lang/StringBuffer/append(D)Ljava/lang/StringBuffer;
56          invokevirtual java/lang/StringBuffer/toString()Ljava/lang/String;
57          invokevirtual java/io/PrintStream/println(Ljava/lang/String;)V
58          return
59  .end method
60
61  .method public static main([Ljava/lang/String;)V
62          .limit stack 11
63          .limit locals 21
64
65  ; instantiate this class and store in local variable 1
66          new BenchmarkPlaceHolder
67          dup
68          invokespecial BenchmarkPlaceHolder/<init>()V
69          astore_1
70
71  ; initialise the parameters to the bench method here (local varibles 2 through 19)
72
73  ; run the benchmark with the above arguments noIterations times
74          iconst_0
75          istore 20
76          goto Condition
77  Body:
78          aload_1
79  ; insert local variable arguments here
80
81          invokevirtual BenchmarkPlaceHolder/bench()V
82          iinc 20 1
83  Condition:
84          iload 20
85          putstatic BenchmarkPlaceHolder/currentIteration I
86          iload 20
87          getstatic BenchmarkPlaceHolder/noIterations I
88          if_icmplt Body
89          return
90  .end method
91
```

Listing 6.5: Part of the place-holder bytecode benchmark class.

that of the bytecode block's original method.

2. For each argument to the '`bench`' method, the '`static void main`' method is instrumented with initialisation bytecode that creates either new instantiations of objects or random arrays, depending on the argument's data type or class. The size of the random arrays is set to the number of times the '`bench`' method is invoked during the course of the benchmark, with each element of the array initialised to a random number between 0 and this number of invocations. This restriction is performed in order to make sure that any variable that may be used as an index to an array is kept within the bounds of the array's size and therefore does not throw an '`ArrayIndexOutOfBounds`' exception. The 'placeholder' class includes a number of methods that create random arrays of specific data types (Listing 6.6 shows a method from the class that creates a random array of doubles). It is important to create these random arrays rather than a constant reference in order to stop the JVM over-optimising a repetitive access in memory to the same reference.

3. Any local variables that are not arguments of the method are initialised within the '`bench`' method prior to executing the benchmark. Each variable is initialised relative to the value of the '`static void main`' method's loop control variable. In order to make sure an exception is not thrown, the bytecode block is interpreted and a record of how the value of each local variable changes, as well as the block's stack access is recorded. This change is then taken into account during the initialisation of the local variables and stack in order to ensure that an exception is not thrown. For example, if a local variable is multiplied by two during an execution of the block, it ensures that the initial value of this variable is below half of the size of all arrays. Any variables that would not throw an exception during the execution of the block are simply initialised to the current value of the loop's control variable.

```
102
103 ; create a random double array of the given size
104 .method public static createRandomDoubleArray(I)[D
105         .limit stack 4
106         .limit locals 3
107
108         iload_0
109         newarray double
110         astore_1
111         iconst_0
112         istore_2
113         goto Condition
114 Body:
115         aload_1
116         iload_2
117         getstatic BenchmarkPlaceHolder/random Ljava/util/Random;
118         invokevirtual java/util/Random/nextDouble()D
119         dastore
120         iinc 2 1
121 Condition:
122         iload_2
123         aload_1
124         arraylength
125         if_icmplt Body
126         aload_1
127         areturn
128 .end method
129
```

Listing 6.6: A method used as part of the benchmark class to randomly populate a double array.

4. During execution, the arguments to a method are put in different local variables, depending on whether the invoked method is static or not. Methods that are static do not include a reference to the method's class instantiation, with the method's arguments starting from local variable 0. To make sure that the block's local variables point to the correct location, a check is made to see if the original method where the block was extracted from was static and, if it was, each local variable referenced by every opcode within the block is incremented by one. References to local variables that are initialised within the benchmark as described above are also incremented.

This ensures that the optimisation achieved after a certain number of executions of the '`bench`' method (once the method count threshold has been exceeded) is as close as possible to the actual in-context optimisation of the bytecode block during the original application's execution.

On finishing the instrumentation of the benchmarking class, the benchmarking

tool executes the class in a separate virtual machine so that the context of the tool itself does not compromise the results. Executing the benchmark results in a nanosecond timing for each execution of the bytecode block. Two timings are then calculated from the results captured from the benchmark's output: the average response time of this bytecode block before and after optimisation. The value declared within the current platform's virtual machine object for the number of iterations of a method before its optimisation is used to calculate these average response times. If this iteration metric was set to 1500, for example, an average of the first 1500 timings output from the benchmark is used as the pre-optimisation time and the average of all the timings after the 1500th timing is used as the optimised time. The total number of iterations used during a benchmark is currently set at 6000, as the experience gained during this research has shown that this allows a significant margin for a reasonable average of optimised results after the method's optimisation.

However, these timings also include the overheads of timing the benchmark in the first place. In order to eliminate these overheads from the results, the instrumented class is again parsed and modified such that the bytecode block being benchmarked is stripped out; all of the initialisation code is left as it is in order to time as accurately as possible the overhead incurred from the rest of the benchmark. This modified benchmark is then executed as before, and the pre- and post-optimisation timings for the no bytecode benchmark calculated as shown previously. The resulting timings are subtracted from the two timings gained from the benchmark including the bytecode block, and it is these final two timings that are used to populate the '`bytecodeBlockTiming`' declaration within the resource, shown in Listing 6.2.

An example of automatically benchmarking a bytecode block and populating the results within the appropriate resource object follows. Listing 6.7 shows an example bytecode block extracted from an MPI-based Sparse Matrix Multiply scientific kernel during its automated characterisation. The benchmark place-holder class shown in Listing 6.5 is parsed and the opcodes that comprise the bytecode block are inserted

between the two timestamps at the beginning and end of the 'bench' method. The descriptor of the 'bench' method is changed to match the original block's method descriptor (from '()V' to '([D[D[I[I[DI[I[D)V'). As local variables 8 and 10 within this method do not represent the method's arguments (the eight arguments of this static method are stored in local variables 0-7 respectively), they are initialised prior to the first time stamp to equal the current iteration of the benchmark. Furthermore, as this method is static, each reference to a local variable is incremented.

```
25
26    <jPACE:bytecodeBlock id="test([D[D[I[I[DI[I[D)V:4" staticMethod="yes">
27
28      <jPACE:OPCODE_aload localVariable="7"/><jPACE:OPCODE_aload_2/>
29      <jPACE:OPCODE_iload localVariable="10"/><jPACE:OPCODE_iaload/>
30      <jPACE:OPCODE_dup2/><jPACE:OPCODE_daload/>
31      <jPACE:OPCODE_aload localVariable="4"/><jPACE:OPCODE_aload_3/>
32      <jPACE:OPCODE_iload localVariable="10"/><jPACE:OPCODE_iaload/>
33      <jPACE:OPCODE_daload/><jPACE:OPCODE_aload_1/>
34      <jPACE:OPCODE_iload localVariable="10"/><jPACE:OPCODE_daload/>
35      <jPACE:OPCODE_dmul/><jPACE:OPCODE_dadd/>
36      <jPACE:OPCODE_dastore/><jPACE:OPCODE_iinc localVariable="10"/>
37      <jPACE:OPCODE_iload localVariable="10"/><jPACE:OPCODE_iload localVariable="8"/>
38      <jPACE:OPCODE_if_icmplt/>
39
40    </jPACE:bytecodeBlock>
41
```

Listing 6.7: The 'test([D[D[I[I[DI[I[D)V:4' bytecode block as stored in the bytecode blocks file during automated characterisation.

The bytecode for the instrumented 'bench' method is shown in Listing 6.8. The bytecode block to be benchmarked can be seen from lines 10-30, between the two timestamps (lines 8 and 31). The two stamps are subtracted at lines 33-36 to calculate the execution time for that iteration of the bytecode block, and this is printed to the standard out on lines 37-44. Local variables 8 and 10 are initialised on lines 4-5 and 6-7 respectively.

The benchmark's 'static void main' method is also instrumented in order to initialise the arguments passed to the 'bench' method, as shown in Listing 6.9. Arguments 1 and 2 are populated with random double arrays and stored in local variables 2 and 3 (lines 55-57 and 58-60 respectively). Arguments 3 and 4 are populated with random integer arrays and stored in local variables 4 and 5 (lines 61-63 and 64-66

```
1
2  ACC_PUBLIC (13) "bench" (43) "([D[D[I[I[DI[I[D)V"
3    ** Attribute Entry 1 (40) "Code"
4       0       getstatic BenchmarkPlaceHolder/currentIteration I
5       3       istore 8
6       5       getstatic BenchmarkPlaceHolder/currentIteration I
7       8       istore 10
8       10      invokestatic NanoTimer/getClockTime()D
9       13      dstore 44
10      15      aload 7
11      17      aload_2
12      18      iload 10
13      20      iaload
14      21      dup2
15      22      daload
16      23      aload 4
17      25      aload_3
18      26      iload 10
19      28      iaload
20      29      daload
21      30      aload_1
22      31      iload 10
23      33      daload
24      34      dmul
25      35      dadd
26      36      dastore
27      37      iinc 10 1
28      40      iload 10
29      42      iload 8
30      44      if_icmplt 3
31      47      invokestatic NanoTimer/getClockTime()D
32      50      dstore 46
33      52      dload 46
34      54      dload 44
35      56      dsub
36      57      dstore 48
37      59      getstatic java/lang/System/out Ljava/io/PrintStream;
38      62      new java/lang/StringBuffer
39      65      dup
40      66      invokespecial java/lang/StringBuffer/<init>()V
41      69      dload 48
42      71      invokevirtual java/lang/StringBuffer/append(D)Ljava/lang/StringBuffer;
43      74      invokevirtual java/lang/StringBuffer/toString()Ljava/lang/String;
44      77      invokevirtual java/io/PrintStream/println(Ljava/lang/String;)V
45      80      return
46
```

Listing 6.8: The instrumented '`static bench`' method used for the benchmarking of the '`test([D[D[I[I[DI[I[D)V:4`' bytecode block

respectively). Argument 5 is populated with another random double array that is stored in local variable 6 (lines 67-69) and argument 6 is initialised to a random integer and stored in local variable 7 (lines 70-73). Finally, arguments 7 and 8 are populated with random integer and double arrays and stored in local variables 8 and 9 (lines 74-76 and 77-79 respectively). During each iteration of the benchmark, these eight arguments are pushed onto the stack (lines 84-91) and the '`invokevirtual`' opcode is used to call

```
46
47  ACC_PUBLIC ACC_STATIC (20) "main" (48) "([Ljava/lang/String;)V"
48    ** Attribute Entry 1 (40) "Code"
49       0       new BenchmarkPlaceHolder
50       3       dup
51       4       invokespecial BenchmarkPlaceHolder/<init>()V
52       7       astore_1
53       8       getstatic BenchmarkPlaceHolder/noIterations I
54       11      istore_1
55       12      iload_1
56       13      invokestatic BenchmarkPlaceHolder/createRandomDoubleArray(I)[D
57       16      astore_2
58       17      iload_1
59       18      invokestatic BenchmarkPlaceHolder/createRandomDoubleArray(I)[D
60       21      astore_3
61       22      iload_1
62       23      invokestatic BenchmarkPlaceHolder/createRandomIntArray(I)[I
63       26      astore 4
64       28      iload_1
65       29      invokestatic BenchmarkPlaceHolder/createRandomIntArray(I)[I
66       32      astore 5
67       34      iload_1
68       35      invokestatic BenchmarkPlaceHolder/createRandomDoubleArray(I)[D
69       38      astore 6
70       40      getstatic BenchmarkPlaceHolder/random Ljava/util/Random;
71       43      iload_1
72       44      invokevirtual java/util/Random/nextInt(I)I
73       47      istore 7
74       49      iload_1
75       50      invokestatic BenchmarkPlaceHolder/createRandomIntArray(I)[I
76       53      astore 8
77       55      iload_1
78       56      invokestatic BenchmarkPlaceHolder/createRandomDoubleArray(I)[D
79       59      astore 9
80       61      iconst_0
81       62      istore 10
82       64      goto 23
83       67      aload_1
84       68      aload_2
85       69      aload_3
86       70      aload 4
87       72      aload 5
88       74      aload 6
89       76      iload 7
90       78      aload 8
91       80      aload 9
92       82      invokevirtual BenchmarkPlaceHolder/bench([D[D[I[I[DI[I[D)V
93       85      iinc 20 1
94       88      iload 20
95       90      putstatic BenchmarkPlaceHolder/currentIteration I
96       93      iload 20
97       95      getstatic BenchmarkPlaceHolder/noIterations I
98       98      if_icmplt -28
99       101     return
100
```

Listing 6.9: The instrumented '`static void main`' method used for the benchmarking of the '`test([D[D[I[I[DI[I[D)V:4`' bytecode block.

the '`bench`' method (line 92).

This benchmark was executed upon two different platforms: '`budweiser`' is a SunUltra V with a 360MHz UltraSparc II processor and 128Mbs RAM running Solaris

```
46
47  ACC_PUBLIC ACC_STATIC (20) "main" (48) "([Ljava/lang/String;)V"
48    ** Attribute Entry 1 (40) "Code"
49       0       new BenchmarkPlaceHolder
50       3       dup
51       4       invokespecial BenchmarkPlaceHolder/<init>()V
52       7       astore_1
53       8       getstatic BenchmarkPlaceHolder/noIterations I
54       11      istore_1
55       12      iload_1
56       13      invokestatic BenchmarkPlaceHolder/createRandomDoubleArray(I)[D
57       16      astore_2
58       17      iload_1
59       18      invokestatic BenchmarkPlaceHolder/createRandomDoubleArray(I)[D
60       21      astore_3
61       22      iload_1
62       23      invokestatic BenchmarkPlaceHolder/createRandomIntArray(I)[I
63       26      astore 4
64       28      iload_1
65       29      invokestatic BenchmarkPlaceHolder/createRandomIntArray(I)[I
66       32      astore 5
67       34      iload_1
68       35      invokestatic BenchmarkPlaceHolder/createRandomDoubleArray(I)[D
69       38      astore 6
70       40      getstatic BenchmarkPlaceHolder/random Ljava/util/Random;
71       43      iload_1
72       44      invokevirtual java/util/Random/nextInt(I)I
73       47      istore 7
74       49      iload_1
75       50      invokestatic BenchmarkPlaceHolder/createRandomIntArray(I)[I
76       53      astore 8
77       55      iload_1
78       56      invokestatic BenchmarkPlaceHolder/createRandomDoubleArray(I)[D
79       59      astore 9
80       61      iconst_0
81       62      istore 10
82       64      goto 23
83       67      aload_1
84       68      aload_2
85       69      aload_3
86       70      aload 4
87       72      aload 5
88       74      aload 6
89       76      iload 7
90       78      aload 8
91       80      aload 9
92       82      invokevirtual BenchmarkPlaceHolder/bench([D[D[I[I[DI[I[D)V
93       85      iinc 20 1
94       88      iload 20
95       90      putstatic BenchmarkPlaceHolder/currentIteration I
96       93      iload 20
97       95      getstatic BenchmarkPlaceHolder/noIterations I
98       98      if_icmplt -28
99       101     return
100
```

Listing 6.9: The instrumented '`static void main`' method used for the benchmarking of the '`test([D[D[I[I[DI[I[D)V:4`' bytecode block.

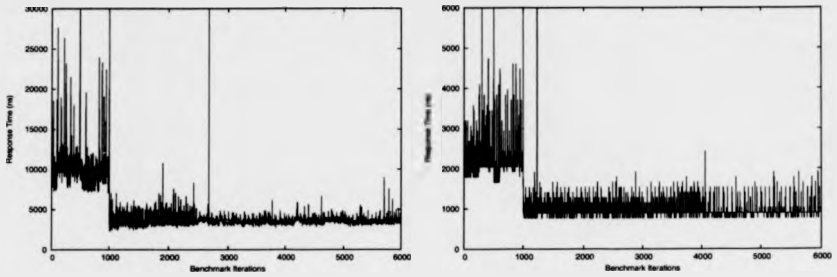
the '`bench`' method (line 92).

This benchmark was executed upon two different platforms: '`budweiser`' is a SunUltra V with a 360MHz UltraSparc II processor and 128Mbs RAM running Solaris

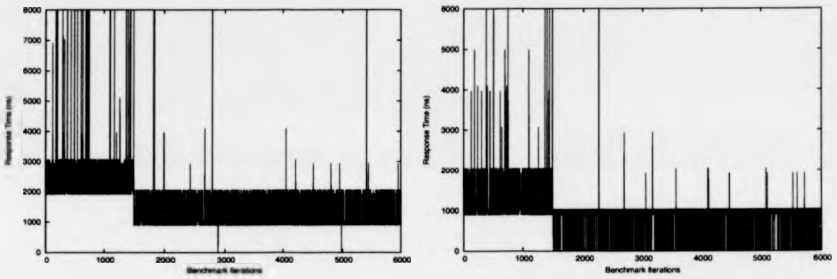
8 and Sun's Hotspot JVM version 1.4.1_01; 'labvista-42' is an IBM Linux workstation with a 800MHz Pentium III processor and 128Mbs RAM running Redhat Linux with kernel 2.4.18-27.7 and Sun's Hotspot JVM version 1.4.1_01. Graphs 6.1 and 6.2 illustrate the variance of the bytecode block's response time over 6000 iterations on these two platforms, as well as the variance in response time of the benchmark without the block present, used to measure the benchmark's overheads. Graph 6.1 shows these benchmark variances for 'budweiser' with and without the bytecode block; Graph 6.2 shows the equivalent benchmark results with and without the bytecode block for 'labvista-42'.

From studying these results, it is clear where the method optimisation threshold lies for each virtual machine. Optimisation takes place at 1000 iterations for 'budweiser' and at 1500 iterations for 'labvista-42', which can be seen from the dramatic decrease in the bytecode block's response time after that point. As stated previously, it is from this observation that the 'runtimeOptimisation' metric within the virtual machine object is set, and the different adaptive compilation strategies used among different virtual machines taken into account during evaluation. Furthermore, it can be seen that the average 'labvista-42' bytecode block and overhead response time is significantly lower that that of 'budweiser'; this greater Java runtime performance is due to the 'labvista-42' workstation's more modern processor architecture and clock speed.

A number of large fluctuations can be seen among these graphs, the majority of which occur prior to optimisation. These fluctuations are the result of the adaptive compilation algorithm making a number of decisions as to whether or not to optimise the bytecode block, as well as the garbage collection algorithm releasing the references to variables that are no longer within the application's scope after each iteration of the method. The response time of these fluctuations is in fact far greater than the maximum y-axis value shown on each graph, but the graph's maximum value was reduced in order to more clearly show the reduction in the block's response time after the block's



Graph 6.1: Benchmark timings for block '`test([D[D[I[I[DI[I[D)V:4`' executing on '`budweiser`': with block (left) and overheads (right).



Graph 6.2: Benchmark timings for block '`test([D[D[I[I[DI[I[D)V:4`' executing on '`labvista-42`': with block (left) and overheads (right).

optimisation.

Upon capturing the two sets of results for each bytecode block being benchmarked upon a given platform, the pre- and post-optimisation response times for the block are calculated, and the '`bytecodeBlockTiming`' declarations within the associated resource object populated accordingly. Table 6.1 shows the captured timings and calculated results for this example, and Listings 6.10 and 6.11 show the eventual populated '`bytecodeBlockTiming`' declarations found within the '`budweiser`' and '`labvista-42`' resource objects respectively.

```
3
4     <jPACE:bytecodeBlockTiming id="test([D[D[I[I[DI[I[D)V:4">
5       <jPACE:iterationTiming noIterations="1.0" executionTime="7802.88"/>
6       <jPACE:iterationTiming noIterations="1000.0" executionTime="7802.88"/>
7       <jPACE:iterationTiming noIterations="1001.0" executionTime="2639.2064"/>
8       <jPACE:iterationTiming noIterations="6000.0" executionTime="2639.2064"/>
9     </jPACE:bytecodeBlockTiming>
10
```

Listing 6.10: The '`test([D[D[I[I[DI[I[D)V:4`' benchmarked resource timings for '`budweiser`'.

```
12
13    <jPACE:bytecodeBlockTiming id="test([D[D[I[I[DI[I[D)V:4">
14      <jPACE:iterationTiming noIterations="1.0" executionTime="1279.829333"/>
15      <jPACE:iterationTiming noIterations="1500.0" executionTime="1279.829333"/>
16      <jPACE:iterationTiming noIterations="1501.0" executionTime="371.1715556"/>
17      <jPACE:iterationTiming noIterations="6000.0" executionTime="371.1715556"/>
18    </jPACE:bytecodeBlockTiming>
19
```

Listing 6.11: The '`test([D[D[I[I[DI[I[D)V:4`' benchmarked resource timings for '`labvista-42`'.

6.3.2 Benchmarking MPI Communication

In the same way that resource objects are populated by benchmarking bytecode block computation response times, MPI domain objects are populated by benchmarking the response times of inter-platform MPI communication. While a resource's '`bytecode BlockTiming`' declarations contain a number of '`iterationTiming`' statements to characterise the changes in response time of a bytecode block during the application's execution, an MPI domain's '`MPITiming`' declaration contains a number of '`commTiming`' statements which characterise the response time of MPI API calls for a range of communication sizes. A predicted response time is then extrapolated from this range of response times for a specific MPI call during evaluation.

Resource	Bytecode Block (ns)		Benchmark Overhead (ns)		Final Result (ns)	
	Unoptimised	Optimised	Unoptimised	Optimised	Unoptimised	Optimised
`budweiser`	10772.864	3586.7904	2969.984	947.584	7802.88	2639.2064
`labvista-42`	2753.109333	1228.401778	1473.28	857.2302222	1279.829333	371.1715556

Table 6.1: This benchmark results for the '`test([D[D[I[I[DI[I[D)V:4`' bytecode block on resources '`budweiser`' and '`labvista-42`.

Unlike benchmarking bytecode blocks, no automated implementation of benchmarking inter-platform communication currently exists. However, there are established methods used in benchmarking this communication that differ slightly, depending on whether the communication is a point-to-point or a collective operation. For a given point-to-point communication between two platforms within a certain MPI Domain, both platforms execute an MPI program where the point-to-point communication API is repeatedly executed for a variety of communication sizes, and the response time of each communication is recorded. An average response time for each communication is calculated and the appropriate MPI domain file is populated accordingly. Listing 6.12 shows the Java source code of an MPI program used in order to benchmark the communication implemented between '`Ssend`' and '`Recv`' MPI APIs.

Benchmarking and characterising collective APIs requires a different approach, since their performance varies not only in the size of the communication but also in the number of platforms and the collective's operation. While it is possible to characterise collective APIs as a series of point-to-point operations (as this is how they are generally implemented), it was decided to characterise a single collective API as a number of separate '`MPITiming`' declarations; one for each separate number of platforms that a collection could be performed on, and one for each separate collective operation that could be chosen. This characterisation does however enforce a restriction, in that it is not possible to characterise a collective API over a range of platforms from different MPI domains; while these platforms themselves can be heterogeneous, they must all be associated with the same MPI domain. Listing 6.13 shows the Java source code of the benchmark used to benchmark the '`Allreduce`' MPI API.

6.4 Summary

This chapter introduced a jPCL platform implementation that characterises the performance of the execution environment of MPI-based Java applications. Five platform

```
32
33      MPI.COMM_WORLD.Barrier();
34
35      if (nprocess == 2) {
36
37          if (rank == 0)
38              System.out.println("Benchmarking ssend MPI.BYTE comms...");
39
40          byte[] sourceArray = new byte[arraySize];
41          byte[] destArray = new byte[arraySize];
42          random.nextBytes(sourceArray);
43          random.nextBytes(destArray);
44
45          double beforeTime, afterTime, totalCommsTime;
46
47          for (int i = 1; i <= arraySize; i *= 2) {
48
49              totalCommsTime = 0;
50
51              for (int j = 0; j < benchmarkIteration; j++) {
52
53                  if (rank == 0) {
54
55                      beforeTime = NanoTimer.getClockTime();
56                      MPI.COMM_WORLD.Ssend(sourceArray, 0, i, MPI.BYTE, 1, 0);
57                      afterTime = NanoTimer.getClockTime();
58
59                      totalCommsTime += (afterTime - beforeTime);
60
61                  }
62
63                  if (rank == 1)
64                      MPI.COMM_WORLD.Recv(destArray, 0, i, MPI.BYTE, 0, 0);
65
66              }
67
68              if (rank == 0)
69                  System.out.println("Ssend MPI.BYTE " + i + " bytes: " +
70                                     (totalCommsTime / benchmarkIteration));
71
72          }
73
74      }
75
76      MPI.COMM_WORLD.Barrier();
77
```

Listing 6.12: A portion of the source code used in benchmarking point-to-point MPI API calls.

interface calls were defined and the details behind their implementation within the Java Hotspot platform were documented. These interface calls are used by the evaluation engine to evaluate the range of a platform's performance that can result during an application's execution. The ability to automate the characterisation of a resource's bytecode computational performance was also described, ensuring that the adaptive compilation of modern JVMs is incorporated within the resource declarations, and a framework for measuring the performance of communication was introduced.

```
160
161     MPI.COMM_WORLD.Barrier();
162
163     if (nprocess >= 2) {
164
165         if (rank == 0)
166             System.out.println("Benchmarking allreduce MPI.DOUBLE MPI.SUM comms...");
167
168         double[] sourceArray = new double[arraySize];
169         double[] destArray = new double[arraySize];
170
171         for (int i = 0; i < arraySize; i++)
172             sourceArray[i] = random.nextDouble();
173
174         double beforeTime, afterTime, totalCommsTime;;
175
176         for (int i = 1; i <= arraySize; i *= 2) {
177
178             totalCommsTime = 0;
179
180             for (int j = 0; j < benchmarkIteration; j++) {
181
182                 beforeTime = NanoTimer.getClockTime();
183                 MPI.COMM_WORLD.Allreduce(sourceArray, 0, destArray, 0, i,
184                                          MPI.DOUBLE, MPI.SUM);
185                 afterTime = NanoTimer.getClockTime();
186
187                 totalCommsTime += (afterTime - beforeTime);
188
189             }
190
191             if (rank == 0)
192                 System.out.println("Allreduce MPI.DOUBLE MPI.SUM " + i +
193                                    " doubles: " +
194                                    (totalCommsTime / benchmarkIteration));
195
196         }
197
198     }
199
```

Listing 6.13: A portion of the source code used in benchmarking collective MPI API calls.

This thesis has now documented how both the performance of an application and the platform's execution environment can be characterised within the jPACE Performance Characterisation Language. A parametric evaluation engine that evaluates these jPCL performance objects has been implemented in order to predict the performance of a characterised application executing on a given platform implementation. This evaluation engine is documented in the following chapter.

```
160
161     MPI.COMM_WORLD.Barrier();
162
163     if (nprocess >= 2) {
164
165         if (rank == 0)
166             System.out.println("Benchmarking allreduce MPI.DOUBLE MPI.SUM comms...");
167
168         double[] sourceArray = new double[arraySize];
169         double[] destArray = new double[arraySize];
170
171         for (int i = 0; i < arraySize; i++)
172             sourceArray[i] = random.nextDouble();
173
174         double beforeTime, afterTime, totalCommsTime;;
175
176         for (int i = 1; i <= arraySize; i *= 2) {
177
178             totalCommsTime = 0;
179
180             for (int j = 0; j < benchmarkIteration; j++) {
181
182                 beforeTime = NanoTimer.getClockTime();
183                 MPI.COMM_WORLD.Allreduce(sourceArray, 0, destArray, 0, i,
184                                          MPI.DOUBLE, MPI.SUM);
185                 afterTime = NanoTimer.getClockTime();
186
187                 totalCommsTime += (afterTime - beforeTime);
188
189             }
190
191             if (rank == 0)
192                 System.out.println("Allreduce MPI.DOUBLE MPI.SUM " + i +
193                                    " doubles: " +
194                                    (totalCommsTime / benchmarkIteration));
195
196         }
197
198     }
199
```

Listing 6.13: A portion of the source code used in benchmarking collective MPI API calls.

This thesis has now documented how both the performance of an application and the platform's execution environment can be characterised within the jPACE Performance Characterisation Language. A parametric evaluation engine that evaluates these jPCL performance objects has been implemented in order to predict the performance of a characterised application executing on a given platform implementation. This evaluation engine is documented in the following chapter.

Chapter 7

The jPACE Evaluation Engine

This thesis has illustrated how to characterise the performance of distributed Java applications in the jPACE Performance Characterisation Language, as well as how to implement a platform interface that models the changes in runtime performance of a Hotspot Java Virtual Machine during an application's execution. This chapter introduces a parametric evaluation engine that evaluates jPCL performance objects in order to calculate a performance prediction. This engine, written in Java for portability, provides a similar command-line interface to the original PACE evaluation engine, enabling the data and scalability analysis of characterised applications.

From the outset, the evaluation engine has been designed to achieve quick evaluations. However, the evaluation of jPCL characterisations is slower than that of PACE performance models. This is because PACE models are compiled executables, whereas during a jPCL evaluation, each XML file is parsed and evaluated on demand. Despite this, a jPCL predictive evaluation normally takes in the order of a few seconds to complete (compared with less than 1 second for a PACE performance model), which is quick enough for a middleware environment to evaluate and make predictive-based decisions.

The evaluation engine is implemented as a stack-based processor. Every application, transaction map and transaction object is an extension of a runtime object

implementation within the evaluation engine. Each runtime object contains:

1. *Runtime Variables*, which contain the runtime object's variables and their values during evaluation;

2. *Runtime Platforms*, which contain the platforms associated with the runtime object during evaluation.

Figure 7.1 illustrates the initialisation of a runtime object. When a new runtime object is initialised, its instance is pushed onto the engine's evaluation stack and popped off once the object's evaluation has concluded. This ensures that the current runtime variables and platforms within the evaluation's current scope are always accessible from the runtime object at the peak of the current evaluation stack.

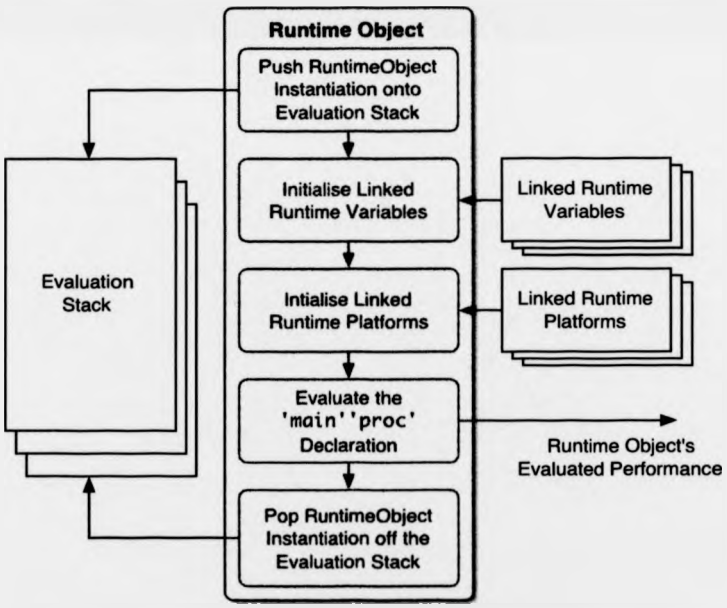


Figure 7.1: The initialisation of a runtime object.

When it is evaluated, each runtime object is passed a number of linked variables. These variables are defined with an object's `'link'` declaration and update the

initial values of an object's runtime variables. The variables linked to an application object are used to update the value of parameters and to define the number of platforms to be evaluated, facilitating the analysis of the characterised application's data and scalability performance. These linked variables can be initialised either by a middleware environment or by the engine's command line interface, enabling the setting of parameter values at the command line.

Upon evaluation, each runtime object is also passed a number of platforms. How these platforms are initialised depends on the type of runtime object. The platforms within an application object are defined by the application's '`platform`' declarations, while platforms within transaction map or transaction objects are defined by the '`evaluateTransactionMap`' and '`evaluateTransaction`' statements respectively.

All mathematical expressions declared within a runtime object are evaluated using a Java expression parser [Funk03], which parses a string expression and calculates its value. Before this calculation is achieved, all variables (whose names are denoted between a '`${`' and '`}`' declaration) are replaced with their current value, as obtained from the current runtime object's runtime variable pool. This ensures that all expressions are accurately calculated during evaluation.

All declarations and statements within a performance model are evaluated to a performance class. The performance class is a simple structure that contains two variables: an evaluated response time, and a confidence associated with the response time. Statements such as '`bytecodeBlock`' and '`MPISsend`', as well as '`method`' declarations of type '`transaction`' or '`native`' are evaluated by accessing the standard platform interface. Every platform declared within the application object must include an implementation of this interface in order to complete an evaluation. This standard platform interface enables the evaluation of a distributed application over a heterogeneous environment containing a variety of differently-performing architectures, since each architecture's characterisation would consist of a different platform

implementation. All other statements and declarations that can be used within a model return an evaluated performance that is a calculation relating to these evaluated performances.

A more detailed description of the evaluation of application, transaction map and transaction objects is given below.

7.1 Evaluating Application Objects

The evaluation engine contains an '`evaluateApplication`' method that is used to evaluate a jPCL application performance object. This method takes two arguments: a file reference to the location of the application object and the application's linked variables. On evaluation, the application object referenced by this file is parsed into an application Java object and pushed onto the engine's runtime stack.

The application's '`platform`' declarations (of which there must be at least one) are then initialised. For each declaration, a new application runtime platform instantiation is created with the name of both the resource and the virtual machine stated within the '`platform`' declaration. This platform instantiation is then initialised and added to the application's runtime pool. During initialisation, both the resource and virtual machine objects, as well as any MPI domain reference declared within the resource, are located within the environment's platform paths (declared within the jPACE configuration file) and parsed as resource, virtual machine and MPI domain Java objects respectively. These platform instantiations are accessed in order to obtain the response times of performance-critical elements during the course of this evaluation.

The linked variables passed as an argument to the object's evaluation are then compared with the application's '`parameter`' declarations. If a linked variable has the same name as that of a '`parameter`' declaration, the parameter's initial value, as defined within the application object, is overwritten with the value of the corresponding link variable. All parameters, whether overwritten or not, are then added to the

application's runtime variable pool as new runtime variable instantiations. An '`nP`' runtime variable is also added to this pool and, unless overwritten by a linked variable, is set to the number of '`platform`' declarations declared within the application object. Finally, all '`variable`' declarations are added to the runtime variable pool as runtime variable instantiations, with their value set to 0 unless otherwise defined since, unlike '`parameter`' declarations, they are not required to state their initial value.

The evaluation engine contains a global variable that specifies the maximum value that can be assigned to confidences. This value is extracted from the application's '`confidence`' declaration so that other performance objects within the model can access them during evaluation.

7.1.1 Evaluating '`proc`' Declarations

Once this initialisation of the application object's runtime variable and platform pools is complete, a '`proc`' declaration called '`main`' is located within the application object. This '`proc`' declaration is defined within jPCL as the entry-point to the runtime object; if a runtime object does not contain this declaration, its evaluated performance is set to a response time of 0 nanoseconds with a maximum confidence. Since the application object is defined as the entry point to the entire model, the performance of this declaration is defined as the evaluated performance of the entire application.

The process of evaluating a '`proc`' declaration is illustrated in Figure 7.2. At the start of evaluation, a '`proc`' performance class is instantiated and initialised to a response time of 0 and a maximum confidence. Each statement within the declaration is then evaluated in turn, and the '`proc`'s' evaluated performance is incremented by each statement's evaluated performance, such that:

$$r_p = r_p + r_s \tag{7.1}$$

$$c_p = \frac{(r_p * c_p) + (r_s * c_s)}{r_p + r_s} \tag{7.2}$$

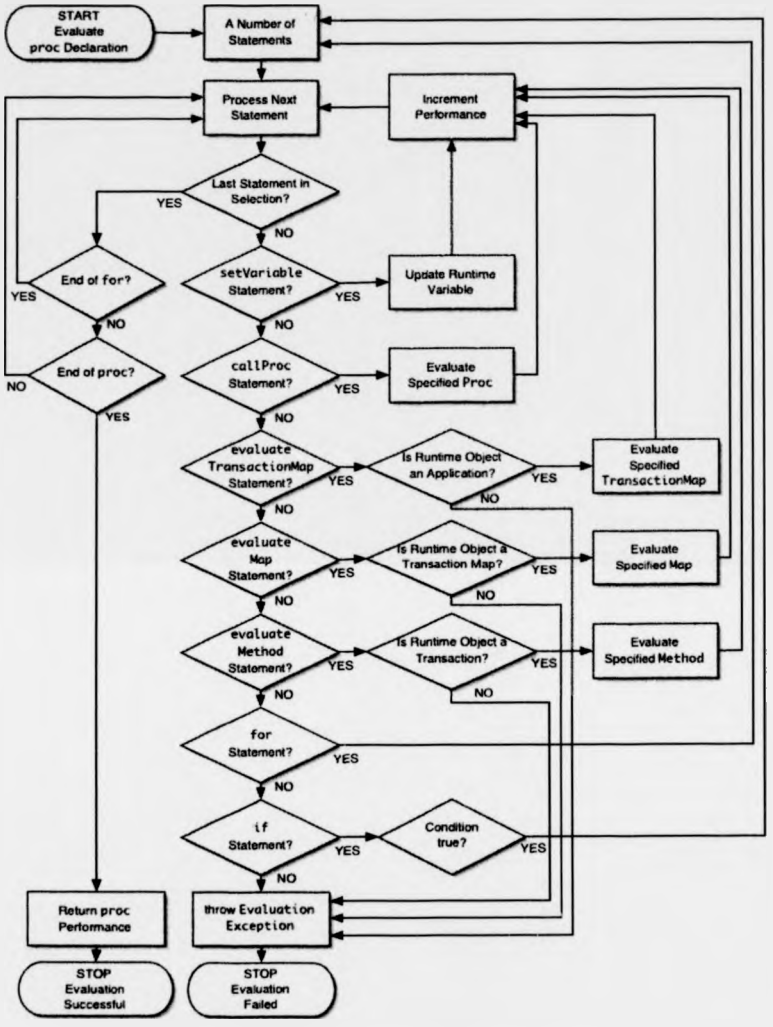


Figure 7.2: Evaluating a '`proc`' declaration.

where r_p and c_p are the values of the '`proc`'s' response time and confidence respectively, and r_s and c_s are the values of the evaluated statement's response time and confidence respectively. This ensures that response times are simply added together, while confidences are weighted according to these response times. An application where

90% of the evaluated response time is associated with a low confidence will, using this method, be assigned a low evaluated confidence.

A '`proc`' declaration can contain any number of the following seven statements[1]:

1. '`setVariable`' statement. Updates a variable within the current runtime object's runtime variables pool. If the variable, whose name is defined by the statement, is located, then its value is set to the value that results from evaluating the statement's value expression. If the variable is not found, an '`Evaluation Exception`' is thrown and evaluation terminates. This statement evaluates to a response time of 0 with a maximum confidence.

2. '`callProc`' statement. Evaluates the specified '`proc`' declaration within this runtime object. If the specified declaration cannot be found within this object, an '`EvaluationException`' is thrown and evaluation terminates. The evaluated performance of this '`proc`' is added to the current '`proc`'s' evaluated performance as defined in Equations 7.1 and 7.2.

3. '`evaluateTransactionMap`' statement. Evaluates the specified transaction map. If this transaction map object cannot be found within the defined model paths, or this '`proc`' declaration being evaluated is not within an application object, an '`EvaluationException`' is thrown and evaluation terminates. As defined within the jPCL specification, transaction maps can only be evaluated from an application object.

   The '`evaluateTransactionMap`' statement must declare which of the runtime platforms currently on the application's runtime platform pool the transaction map should be evaluated upon. This platform declaration can be the codeword '`ALL`', meaning the transaction map is evaluated on all of the application's runtime platforms, a constant platform index that references a single platform

[1] these statements apply to a '`proc`' declaration within any runtime object

within the runtime platform pool, or a range of platform indexes denoted by '--'. For example, the platform declaration '1 -- ${nP} - 1' states that the transaction map is to be evaluated on the platforms with an index of between 1 and the number of platforms minus 1 inclusive.

Prior to the transaction map's evaluation, two pools of linked runtime variables and runtime platforms are created. The variable pool is populated with the name and calculated value of any 'link' declarations whose target object is this transaction map. The platform pool is populated with the platforms whose indexes are declared within the 'evaluateTransactionMap' statement, as described above. The transaction map's file reference, the variable pool and the platform pool are then passed as arguments to the engine's 'evaluateTransaction Map' method. The evaluated performance of this transaction map is then added to the current 'proc's' evaluated performance as previously defined.

4. 'evaluateMap' statement. Evaluates the specified 'map' declaration within the same runtime object. If the specified declaration cannot be found within the same object, or the 'proc' declaration currently being evaluated is not within a transaction map object, an 'EvaluationException' is thrown and evaluation terminates. As defined within the jPCL specification, 'map' declarations can only be evaluated from a transaction map object. The evaluated performance of this 'map' is added to the current 'proc's' evaluated performance as previously defined.

5. 'evaluateMethod' statement. Evaluates the specified 'method' declaration within this runtime object. If the specified declaration cannot be found within this object, or this 'proc' declaration is not within a transaction object, an 'EvaluationException' is thrown and evaluation terminates. As defined within the jPCL specification, 'method' declarations can only be evaluated from a transaction object. The evaluated performance of this 'method' is

added to the current '`proc`'s' evaluated performance as previously described.

6. '`for`' statement. Repeatedly evaluates a specified number of statements for a defined number of iterations. The '`for`' statement defines a '`variable`' name that is used as the '`for`' statement's control variable. A new runtime variable with this name is added to the runtime object's variable pool, and its initial value is set to the evaluated value of the '`for`' statement's '`startValue`' attribute. Each statement defined within the '`for`' statement is then repeatedly evaluated, with each statement's evaluated performance added to the current '`for`' statement's evaluated performance. After each iteration, the control variable is set to the evaluated value of the '`increment`' attribute, and iteration continues only if the control variable has not exceeded the evaluated value of the '`endValue`' attribute. Once the control variable has exceeded this value, the control variable is removed from the runtime variable pool and the '`for`' statement's evaluated performance is added to the current '`proc`'s' evaluated performance.

7. '`if`' statement. Evaluates the specified number of statements if the defined condition is true at the time of the statement's evaluation. If the condition is met, each statement is evaluated once, and the '`if`' statement's evaluated performance is the total of these statements' evaluated performances. If the condition is not met, the statement is evaluated to have a response time of 0 and an associated maximum confidence. The evaluated performance of this '`if`' statement is added to the current '`proc`'s' evaluated performance.

Once all of the statements within the application object's entry '`proc`' declaration have been evaluated, the instantiation of this application object is popped off the engine's runtime stack and the evaluation terminates.

7.2 Evaluating Transaction Map Objects

The evaluation engine contains an '`evaluateTransactionMap`' method that is used to evaluate a jPCL transaction map performance object. This method takes three arguments: a file reference to the location of the transaction map object, the transaction map's linked variables and the transaction map's linked platforms. On evaluation, the transaction map object referenced by this file is parsed into a transaction map Java object and pushed onto the engine's runtime stack. The transaction map's runtime platform pool is populated with the same platform instantiations that were passed from the application object.

The transaction map's runtime variable pool is then populated with a new instantiation of runtime variable for each transaction map '`variable`' declaration. Each variable's initial value is modified to the linked variable's value if their names are the same. If no value is assigned to any of these runtime variables, either by the linked variables or from within the transaction map's '`variable`' declarations, their value is set to 0. The 'number of platforms' runtime variable '`nP`' is also added to the transaction map's runtime variable pool with its value set to the number of platforms held within the transaction map's runtime platforms pool.

Once this initialisation of the transaction map's runtime variable and platform pools is complete, a '`proc`' declaration called '`main`' is located within the transaction map object. Each statement within this '`proc`' is evaluated as documented previously, and the total of their evaluated performances is defined as the transaction map's evaluated performance.

7.2.1 Evaluating '`map`' Declarations

Transaction maps contain '`map`' declarations that characterise the relationship between an application's transactions. The process of evaluating a '`map`' declaration is illustrated in Figure 7.3. At the start of evaluation, a '`map`' performance class is instanti-

ated and initialised to a response time of 0 and a maximum confidence. Each statement within the declaration is evaluated in turn, and the 'map's' evaluated performance is incremented by each statement's evaluated performance.

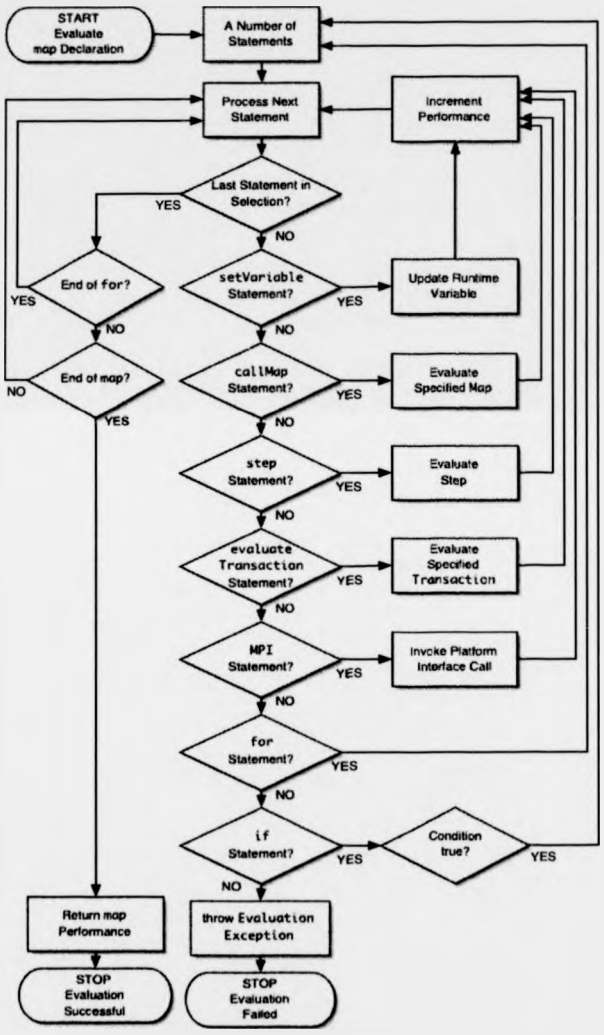


Figure 7.3: Evaluating a 'map' declaration.

'map' declarations contain a control flow of 'step' declarations that characterise performance-critical elements executing concurrently on either a single processor or different platforms within a distributed environment. It is assumed during evaluation that all of the performance-critical elements defined within a single 'step' wait for the other elements to finish executing (as if an 'MPI Barrier' statement is used, for example). Each statement within the declaration is evaluated in turn, and the evaluated performance of the 'step' declaration is set to whichever statement's performance contains the largest response time.

A 'step' declaration must contain at least one of the following statements in order to evaluate an application's performance-critical elements:

1. 'evaluateTransaction' statement. Evaluates the specified transaction. If this object cannot be found within the defined model paths, an 'Evaluation Exception' is thrown and evaluation terminates.

   Transactions are initialised and evaluated from a 'map' declaration in much the same way as transaction maps are initialised and evaluated from an application's 'proc' declaration. Prior to the transaction's evaluation, a pool of runtime variables and a pool of runtime platforms are created. The variable pool is populated as defined by the runtime object's 'link' declarations, and the platform pool is populated with references to the transaction map's runtime platform pool as defined by the 'evaluateTransaction' statement. The transaction's file reference, the variable pool and the platform pool are then passed as arguments to the engine's 'evaluateTransaction' method, which returns an evaluated performance instantiation for the transaction.

2. 'MPI' statement. Evaluates the specified MPI inter-platform communication. Each 'MPI' statement (such as 'MPISsend') defines the source and destination MPI API, the source and destination platform, and the size of the communication. An evaluated response time for point-to-point and collective communi-

cations is obtained from the '`getMPIDomainPToPResponseTime()`' and '`getMPIDomainCollectiveResponseTime()`' platform interface calls respectively. The confidence associated with this response time is set to the maximum confidence value as all information obtained from benchmarks, including benchmarked communication timings, are trusted within the jPACE framework.

When all the statements within the transaction map's entry level '`proc`' declaration have been evaluated, the transaction map instantiation is popped off the engine's runtime stack and its evaluation terminates.

7.3 Evaluating Transaction Objects

The evaluation engine contains an '`evaluateTransaction`' method, which is used to evaluate a jPCL transaction performance object. This method takes three arguments: a file reference to the location of the transaction object, the transaction's linked variables and the transaction's linked platforms. On evaluation, the transaction object referenced by this file is parsed into a transaction Java object and pushed onto the engine's runtime stack. The transaction's runtime platform pool is populated with the same platform instantiations that were passed from the transaction map object.

The transaction's runtime variable pool is then populated with a new instantiation of runtime variable for each transaction '`variable`' declaration. Each variable's initial value is modified to the linked variable's value if their names are the same. If no value is assigned to any of these runtime variables, either by the linked variables or from within the transaction map's '`variable`' declarations, their value is set to 0. The 'number of platforms' runtime variable '`nP`' is also added to the transaction's runtime variable pool with its value set to the number of platforms held within the transaction's runtime platforms pool.

Once this initialisation of the transaction's runtime variable and platform pools is complete, a '`proc`' declaration called '`main`' is located within the transaction ob-

ject and evaluated for every platform within the current runtime platform pool. This enables the characterisation of distributed kernels as one transaction and reduces the time taken to create the performance model. Prior to evaluating this '`proc`' declaration, a runtime variable called '`cP`', which holds the index of the current platform within the transaction's runtime platforms pool being evaluated, is added to the transaction's runtime variables pool. This enables a transaction's evaluation to be controlled according to the current platform. Each statement within this '`proc`' is evaluated as documented previously, and the total of their evaluated performances is defined as the transaction's performance upon the given platform. The evaluated performance with the longest response time is used as the final evaluated performance of this transaction object.

7.3.1 Evaluating '`method`' Declarations

Transactions characterise performance-critical items of work within a '`method`' declaration. The process of evaluating a '`method`' declaration is illustrated in Figure 7.4.

During the course of a '`method`' evaluation, any un-trusted characterisations from either data-dependent areas of an application or from evaluating a response time from historical data are associated with a confidence. The value of this confidence is obtained by evaluating the transaction's '`confidence`' declaration. Prior to evaluating this declaration, the number of times the current method has previously been executed is obtained by calling the current platform's '`getMethodNoExecutions()`' platform interface call. This value is added as a runtime variable to the transaction's runtime variable pool with the name '`nE`'. A confidence variable, whose name is defined by the '`confidence`' declaration, is also added to this pool with its initial value set to 0. Each statement in the declaration is evaluated in turn, and the final value of this confidence value is used as the evaluated confidence of this un-trusted characterisation. The confidence variable and the runtime variable named '`nE`' is finally removed from

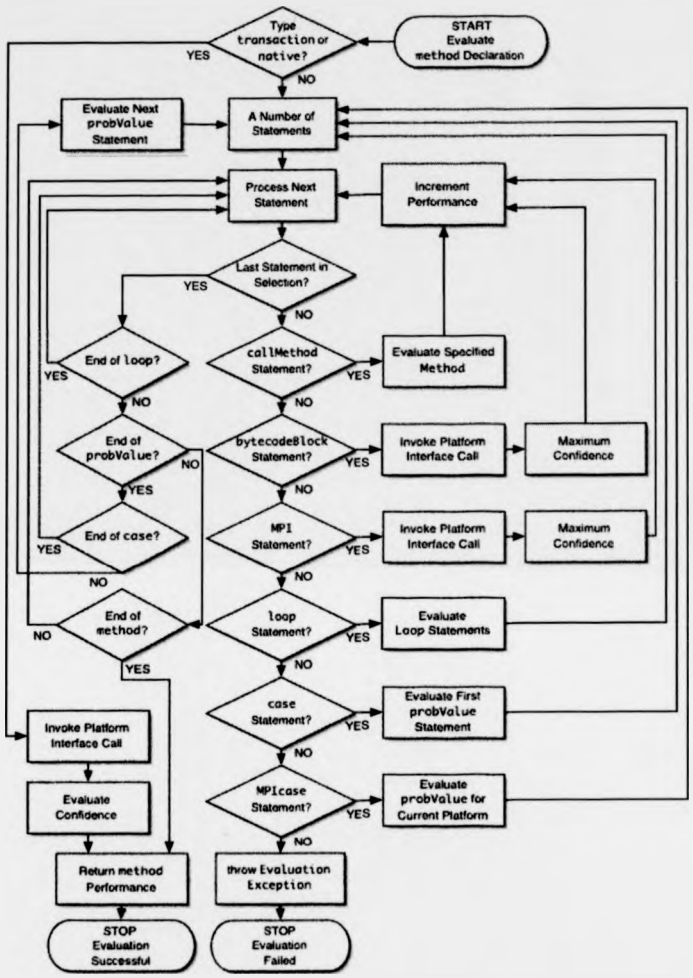


Figure 7.4: Evaluating a '`method`' declaration.

the runtime pool.

At the start of evaluation, a '`method`' performance class is instantiated and initialised to a response time of 0 and a maximum confidence. If the '`method`' declaration is either of type '`transaction`' or '`native`', the response time of this performance class is set to the result of calling the current platform's '`getMethodAv`

`ResponseTime()`' platform interface call. The transaction's '`confidence`' declaration is then evaluated for this method, and the result is set to the confidence of the '`method`'s' performance. The resulting performance class is returned as the evaluated performance of this '`method`' on the current runtime platform.

The evaluation of a '`method`' declaration of type '`characterised`' is more complex. Such a declaration is composed of any number of the following six statements:

1. '`bytecodeBlock`' statement. Characterises blocks of bytecode within a method. The response time of the statement is evaluated from the current platform's '`getBytecodeBlockResponseTime()`' platform interface call, and the statement's confidence is set to its maximum value. Benchmarked timings, within the jPACE framework, are assumed 100% accurate during evaluation.

2. '`MPI`' statement. Characterises an inter-platform MPI communication. As with MPI statements used within a transaction map's '`step`' declaration, these statements are evaluated by invoking the '`getMPIDomainPToPResponseTime()`' and '`getMPIDomainCollectiveResponseTime()`' platform interface calls. However, because transactions are only evaluated for one platform at a time, MPI calls within a transaction are restricted in that their source and destination platform indexes are not declared. This helps the automated characterisation tool to characterise these MPI method calls. An MPI statement's response time is therefore calculated as the average response time between the current platform and all other platforms within the transaction's runtime platforms pool. The evaluated confidence is set to its maximum value, as MPI timings are achieved from benchmarks. If required, further specific communication characterisation can be performed within a transaction map's '`map`' declaration.

3. '`loop`' statement. Characterises iterative areas of bytecode. During its evaluation, each '`evaluateMethod`' statement keeps a current loop count variable

(called '`cLC`' for clarification), which is initially set to 1. Every time a '`loop`' statement is evaluated, '`cLC`' is multiplied by the evaluated loop count of the '`loop`' statement until the completion of the statement's evaluation, when it is returned to its original value. Any '`getBytecodeBlockResponseTime()`' interface calls made during the '`loop`' statement's evaluation include the current value of '`cLC`', and the platform implementation takes this loop count into account when evaluating a bytecode block, as documented in the previous chapter. Any other platform interface calls do not include this '`cLC`' loop count, and so the evaluation engine multiplies the statement's response time obtained from a interface call by this loop count. The evaluated performance of the '`loop`' statement is calculated as the total evaluated performance of each of the statements within the '`loop`' statement, with the value of '`cLC`' taken into account. This method of evaluating iterative areas of an application means that each statement within a '`method`' declaration is evaluated just once, thus greatly reducing the overall evaluation time.

If the '`loop`' statement is declared as data-dependent, then the evaluated confidence of this statement is modified in relation to the transaction's '`confidence`' declaration. The final evaluated confidence ranges linearly from the original total evaluated confidence (if the '`confidence`' declaration evaluates to the maximum value) to the minimum confidence value of 0. Mathematically, the final evaluated confidence is calculated as:

$$c_e = \frac{c_{LOOP} * c_{TRAN}}{c_{MAX}} \tag{7.3}$$

where c_{LOOP} is the original total evaluated confidence of all of the statements defined within the '`loop`' statement, c_{MAX} is the maximum confidence value declared within the model's application object, c_{TRAN} is the evaluated confidence of this method as defined within the transaction's '`confidence`' declaration, and c_e is the eventual confidence assigned to the '`loop`' statement's evaluated

performance. Assigning the confidence in this way ensures that data-dependent expressions within a transaction result in a confidence relative to that declared within the '`confidence`' declaration.

4. '`case`' statement. Characterises conditional areas of bytecode. Each '`case`' statement contains a number of '`probValue`' statements which characterise the probability of an area of bytecode executing at a specific time during an application's execution. Each '`evaluateMethod`' statement keeps a current probability value variable (called '`cPV`' for clarification) during its evaluation, which is set to 1 initially. On a '`case`' statement's evaluation, each '`probValue`' statement is evaluated with this value of '`cPV`' multiplied by the evaluated probability value of the '`probValue`' statement, until the conclusion of this statement's evaluation when it is reset back to its original value. Any '`getBytecode BlockResponseTime()`' interface calls made during the '`probValue`' statement's evaluation include the current value of '`cPV`', and the platform implementation takes this probability value into account when evaluating a bytecode block. The evaluation engine multiplies the result of any other platform interface call by this current probability value. The evaluated performance of this '`case`' statement is the total evaluated performance of every '`probValue`' statement. The evaluated performance of a '`probValue`' statement is the total performance of each of the statements within the '`probValue`' statement, with the value of '`cPV`' taken into account. If, however, the '`probValue`' is declared as data-dependent, its evaluated confidence is modified as documented for data-dependent '`loop`' statements.

5. '`MPIcase`' statement. A '`MPIcase`' statement characterises those areas of bytecode within a distributed application whose execution is dependent upon the current platform. Each '`MPIcase`' statement contains a number of '`probValue`' statements that declare which platform the area of bytecode will execute upon

during the application's execution. Evaluating an '`MPIcase`' statement is achieved by evaluating the '`probValue`' statement that declares the same platform index as the value of the current platform index variable '`cP`'. The evaluated performance of a '`probValue`' statement is the same as that documented previously for a '`case`' statement. '`probValue`' statements within an '`MPIcase`' statement cannot, by definition, be declared data-dependent.

6. '`callMethod`' statement. A '`callMethod`' statement characterises the invocation of another method. The '`method`' declaration with the same method class, name and descriptor as that which is defined within the '`callMethod`' statement is evaluated in the same way as this '`method`' declaration, except that the current loop count ('`cLC`') and probability value ('`cPV`') variables are left at their current value. The evaluated performance of a '`callMethod`' statement is therefore the total calculated performance of all the statements declared within the '`method`' declaration.

Once all of the statements within the transaction's entry level '`proc`' declaration have been evaluated, this instantiation of the transaction map is popped off the engine's runtime stack and the transaction's evaluation terminates.

7.4 Summary

This chapter introduced a parametric evaluation engine that is used to obtain predictions from jPCL performance characterisations. Evaluated predictions can be achieved from either a platform's benchmark timings or historical data obtained during the application's execution, with a confidence associated with all non-trusted performance data as defined within the '`confidence`' declaration. Furthermore, an interface has been implemented that allows access to the evaluation engine on the command-line and, since it is implemented in Java, from within any Java application.

Evaluations vary in length depending on the size of the model, the number of platforms the model is evaluated upon and the resource used to perform the evaluation. The time taken to evaluate the five scientific kernels that are characterised in this research on a Linux workstation containing an Intel Pentium IV 2GHz processor varied between 1 and 10 seconds.

The following two chapters document the results obtained from predicting the performance of Java applications using the predictive framework described so far within this thesis. The performance of five scientific kernels are characterised using jPCL, and bytecode block and MPI communication timings are benchmarked on three differently-performing platforms, as described previously. It is shown that the majority of predictive results obtained from using this framework are within 20% of the actual, measured response time.

Chapter 8

The Performance Prediction of Scientific Kernels

This thesis has introduced a number of extensions to the PACE toolkit in order to create a more dynamic, flexible and portable prediction framework for distributed applications. These extensions include: a newly developed language for the performance description of distributed Java applications named the jPACE Performance Characterisation Language (jPCL); a platform interface implementation of a Hotspot Java Virtual Machine, which models the runtime optimisations of Java execution environments; an evaluation engine that calculates performance predictions of characterised applications executing upon modelled platforms; a number of tools that automate this process in order to facilitate rapid performance predictions. The previous chapters have documented the implementation of these extensions within the newly developed jPACE framework.

This thesis also documents the performance characterisation and evaluation of five scientific kernels using the jPACE framework. These kernels are taken from the JavaGrande benchmark suite [Bull00], a popular resource within the high-performance community for evaluating the performance of Java-based scientific applications. The kernels chosen include MPI-based implementations of a Sparse Matrix Multiply, Fourier Coefficient Analysis and the International Data Encyption Algorithm (IDEA), as well

as two sequential Java implementations of the NAS Gaussian Random Number Generator and Fast Fourier Transform (FFT) benchmarks, developed at the University of Adelaide [Mathew99]. The performance-critical kernel of each application is characterised in jPCL and evaluated on a number of differently-performing platforms in order to predict its performance prior to execution; any initialisation of data or final verification is not characterised within these experiments. The application's defined bytecode blocks and MPI communication APIs are benchmarked prior to evaluation and used to populate the Hotspot JVM platform implementation's resource and MPI domain objects respectively. The predicted performance is compared with the measured performance in order to calculate the predictive accuracy that can be achieved when using the jPACE framework.

The prediction of these applications is documented in both this and the following chapter. The three MPI-based applications are completely parameter-dependent, enabling accurate predictive results to be achieved prior to the application's execution[1]. The prediction of these three applications is discussed in this chapter, with a focus on the performance model's development, its evaluation, and the data and scalability analysis of the applications on a number of platforms.

In contrast, the two sequential benchmarks contain a number of data-dependent areas of code, which require historical data in order to achieve accurate predictions. The following chapter describes a performance-monitoring framework for Java applications that is used to automate the refinement of these data-dependent characterisations within a performance model. It is shown that an application's predictive evaluation becomes more accurate as more historical data is gathered and its associated confidence grows closer to its maximum value. The prediction of these two benchmarks is discussed, with a different focus employed in order to illustrate this automated refinement.

This chapter consists of three sections, which document the Sparse Matrix Mul-

[1] a number of calls to the '`java.lang.Math`' and '`java.util.Random`' API are benchmarked in order for this to be achieved

tiply, Fourier Coefficient Analysis and IDEA Encryption kernels respectively. Predictive evaluations are verified with the measured execution performance of these kernels on a cluster of 16 Redhat Linux workstations (named '`mscs-01`' to '`mscs-16`'), each containing a 2.4GHz Intel Pentium IV processor and 512 MBs RAM, connected by a 100Mb Ethernet network. Verification of predictive evaluations on other architectures is documented in the following chapter.

8.1 Sparse Matrix Multiply

The Sparse Matrix Multiply JavaGrande benchmark is an MPI implementation of the equivalent sequential Scimark benchmark that calculates the function '$y = Ax$'. 'A' is an unstructured sparse matrix of size '`N x N`', stored in compressed-row format with a prescribed sparsity structure of '`nz`' non-zero values. 'y' is a '`M x 1`' vector and 'x' is a '`1 x N`' vector. '`M`', '`N`' and '`nz`' are parameters, where '`M`' must equal '`N`' for all benchmark executions.

In order to parallelise the benchmark, the three arrays in which the sparse matrix 'A' is encoded are divided roughly equally among the available processors. The matrix is created and initialised by the master processor and then sent to the other processors. Each processor contains a copy of the random vector 'x' and calculates the result of 'y' for its section of 'A' for a predefined number of iterations. After this number of iterations, a collective communication updates the global copy of 'y', which is verified by the master processor at the conclusion of the benchmark.

The performance-critical element of this benchmark is implemented as two methods: the '`JGFinitialise`' method that initialises the matrix data and copies it to the slave processors, and the '`JGFkernel`' method that performs the multiplication and updates the result in the other processors' copy of 'y'. The full source code of this benchmark is shown in Appendix A.

8.1.1 Characterising the Application

While developing the model, the choice was made to characterise the benchmark as two transactions: one each for the '`JGFinitialise`' and '`JGFkernel`' methods. Each transaction would characterise both the method's bytecode and its communication. Since the target platform is a cluster of workstations within the same MPI domain, the evaluated communication between any processors will remain accurate. The model's transaction map will simply evaluate these two transactions in turn on all of the model's specified platforms.

The full performance characterisation of this benchmark is shown in Appendix B. The benchmark's application object defines sixteen '`platform`' declarations in order to evaluate the object on the 16-node '`mscs`' cluster; the number of machines that the model is actually evaluated upon can be set prior to evaluation by setting the jPCL '`nP`' parameter. Three '`parameter`' declarations are defined in order to parameterise the model with the benchmark parameters '`M`', '`N`' and '`nz`', with their initial values set to one of the defined benchmark class sizes. Three '`link`' declarations are used to link parameter values to the transaction map object called '`smm.tranmap`', which is evaluated by the application's entry '`proc`' declaration. Being the only statement within the application's entry '`proc`', the evaluated response time of this transaction map is defined as the evaluated performance of the entire model. Furthermore, the maximum confidence value that can be applied to evaluated performances is set to 1.

The model's '`smm.tranmap`' transaction map contains a single '`map`' declaration that defines two synchronous steps. The first '`step`' declaration evaluates the '`JGFinitialise`' method's transaction characterisation (named '`smm-init.tran`') and the second evaluates the '`JGFkernel`' method's transaction (named '`smm-kernel.tran`'). Both transactions are evaluated on all platforms. The transaction map's entry '`proc`' declaration contains one statement that evaluates this map, and so the total evaluated performance of these two transactions will result in the entire model's evaluated performance. The three parameters linked from the application

object are passed on to these transactions as required.

During development, the '`smm-init.tran`' transaction was initially populated with a '`method`' declaration that referenced the original '`JGFinitialise`' benchmark method, as well as three '`variable`' declarations called '`nprocess`', '`N`' and '`p_datasizes_nz`'. The automated characterisation tool (ACT) was then executed on this transaction in order to populate its declaration with the correct jPCL statements. Five '`method`' declarations resulted: three of type '`transaction`' that characterise the '`java.lang.Math`' and '`java.util.Random`' API methods used to initialise the matrix data, and two of type '`characterised`' for the benchmark's '`RandomVector`' and '`JGFinitialise`' methods. In order to simplify the model, an option had been set in the jPACE configuration file to characterise all '`java.*`' API calls as type transaction.

Declaring the three variables prior to this automated characterisation resulted in a number of the method's '`loop`' statements being populated with the correct parameter-dependent expressions. Due to the fact that extra debugging information was turned on during the benchmark's compilation, the ACT was able to match the variables used within the original method with the transaction's '`variable`' declarations. The '`nprocess`' variable was used to enable the ACT to match the benchmark's 'number of processors' variable. After executing the ACT, this '`variable`' declaration was removed from the transaction, and any occurrences of '`nprocess`' within the transaction were modified to the standard jPCL '`nP`' variable. Listing 8.1 contains a comparison of the original benchmark source code and its characterised '`loop`' declarations.

The four conditional statements within the '`JGFinitialise`' method that check the current rank of the processor could not be calculated automatically by the ACT but were instead defined manually. These expressions were simple to calculate, and all the '`case`' statements defined by the ACT were modified to '`MPIcase`' statements in order to characterise the change of behaviour that exists among the processors.

```
103
104             for (int i=0; i<p_datasizes_nz; i++) {
105
106                 // generate random row index (0, M-1)
107                 row[i] = Math.abs(R.nextInt()) % datasizes_M[size];
108                 buf_row[i] = row[i];
109                 // generate random column index (0, N-1)
110                 col[i] = Math.abs(R.nextInt()) % datasizes_N[size];
111                 buf_col[i] = col[i];
112                 val[i] = R.nextDouble();
113                 buf_val[i] = val[i];
114             }
115
116             for(int k=1;k<nprocess;k++) {
117
118                 if(k==nprocess-1) {
119                     p_datasizes_nz = rem_p_datasizes_nz;
120                 }
121
```

```
95
96          <jPACE:loop count="${p_datasizes_nz}">
97
98            <jPACE:bytecodeBlock id="JGFinitialise()V:8"/>
99            <jPACE:callMethod class="java.util.Random"
100                              method="nextInt" descriptor="()I"/>
101           <jPACE:bytecodeBlock id="JGFinitialise()V:9"/>
102           <jPACE:callMethod class="java.lang.Math"
103                              method="abs" descriptor="(I)I"/>
104           <jPACE:bytecodeBlock id="JGFinitialise()V:10"/>
105           <jPACE:callMethod class="java.util.Random"
106                              method="nextInt" descriptor="()I"/>
107           <jPACE:bytecodeBlock id="JGFinitialise()V:11"/>
108           <jPACE:callMethod class="java.lang.Math"
109                              method="abs" descriptor="(I)I"/>
110           <jPACE:bytecodeBlock id="JGFinitialise()V:12"/>
111           <jPACE:callMethod class="java.util.Random"
112                              method="nextDouble" descriptor="()D"/>
113           <jPACE:bytecodeBlock id="JGFinitialise()V:13"/>
114
115         </jPACE:loop>
116
117         <jPACE:bytecodeBlock id="JGFinitialise()V:14"/>
118
119         <jPACE:loop count="${nP} - 1">
120
121           <jPACE:bytecodeBlock id="JGFinitialise()V:15"/>
122
123           <jPACE:case>
124             <jPACE:probValue value="1 / (${nP} - 1)">
125
126               <jPACE:bytecodeBlock id="JGFinitialise()V:16"/>
127
128             </jPACE:probValue>
129           </jPACE:case>
130
```

Listing 8.1: A section of the original '`JGFinitialse`' method (top) and its characterised jPCL statement (bottom).

Listing 8.2 illustrates an example portion of the benchmark that is just executed on the master processor and its associated jPCL '`MPIcase`' statement.

The MPI communication APIs were also automatically characterised by the

```
91
92          if(rank==0) {
93              buf_val = new double[datasizes_nz[size]];
94              buf_col = new int[datasizes_nz[size]];
95              buf_row = new int[datasizes_nz[size]];
96          }
97
```

```
80
81      <jPACE:MPIcase>
82        <jPACE:probValue platform="0">
83
84          <jPACE:bytecodeBlock id="JGFinitialise()V:5"/>
85
86        </jPACE:probValue>
87      </jPACE:MPIcase>
88
```

Listing 8.2: An original '`JGFinitialise`' method's conditional statement (top) and its characterised jPCL '`MPIcase`' statement (bottom).

ACT. Each MPI communication from the original benchmark defines its size as the value of '`p_datasizes_nz`', so this was used within the transaction's characterisation. The communication's data type was also declared in the original bytecode and defined within the jPCL '`MPI`' statement. Listing 8.3 shows an example original MPI communication and the resulting jPCL characterisation.

```
128
129     MPI.COMM_WORLD.Ssend(buf_row,(k*ref_p_datasizes_nz),p_datasizes_nz,MPI.INT,k,1);
130     MPI.COMM_WORLD.Ssend(buf_col,(k*ref_p_datasizes_nz),p_datasizes_nz,MPI.INT,k,2);
131     MPI.COMM_WORLD.Ssend(buf_val,(k*ref_p_datasizes_nz),p_datasizes_nz,MPI.DOUBLE,k,3);
132
```

```
153
154     <jPACE:bytecodeBlock id="JGFinitialise()V:24"/>
155     <jPACE:MPISsend destAPI="Recv" datatype="MPI.INT" size="${p_datasizes_nz}"/>
156     <jPACE:bytecodeBlock id="JGFinitialise()V:25"/>
157     <jPACE:MPISsend destAPI="Recv" datatype="MPI.INT" size="${p_datasizes_nz}"/>
158     <jPACE:bytecodeBlock id="JGFinitialise()V:26"/>
159     <jPACE:MPISsend destAPI="Recv" datatype="MPI.DOUBLE" size="${p_datasizes_nz}"/>
160     <jPACE:bytecodeBlock id="JGFinitialise()V:27"/>
161
```

Listing 8.3: The original '`JGFinitialise`' MPI communication APIs (top), and its associated jPCL characterisation (bottom).

While the '`smm-init.tran`' transaction's '`N`' and '`nz`' '`variable`' declarations were linked from the model's transaction map, variable '`p_datasizes_nz`' needed to be initialised before the '`method`' declaration could be evaluated. This was achieved within the transaction's entry '`proc`' declaration prior to the '`evaluate`

Method' statement. Listing 8.4 shows the original code where this variable is set, and the resulting jPCL characterisation that initialises the 'p_datasizes_nz' variable.

```
70
71          p_datasizes_nz = (datasizes_nz[size] + nprocess - 1) / nprocess;
72          ref_p_datasizes_nz = p_datasizes_nz;
73          rem_p_datasizes_nz = p_datasizes_nz -
74              ((p_datasizes_nz*nprocess) - datasizes_nz[size]);
75
76          if(rank==(nprocess-1)){
77              if((p_datasizes_nz*(rank+1)) > datasizes_nz[size]) {
78                  p_datasizes_nz = rem_p_datasizes_nz;
79              }
80          }
81
```

```
12
13    <jPACE:proc name="main">
14      <jPACE:setVariable variable="p_datasizes_nz"
15                         value="(${nz} + ${nP} - 1) / ${nP}"/>
16
17      <jPACE:if leftExpression="${cP}" condition="EQUALS" rightExpression="${nP} - 1">
18        <jPACE:if leftExpression="${p_datasizes_nz}*${nP}" condition="GREATER_THAN"
19                  rightExpression="${nz}">
20          <jPACE:setVariable variable="p_datasizes_nz"
21                             value="${p_datasizes_nz} -
22                                    ((${p_datasizes_nz}*${nP}) - ${nz})"/>
23        </jPACE:if>
24      </jPACE:if>
25
26      <jPACE:evaluateMethod
27         class="uk.ac.warwick.dcs.hpsg.applications.jgf.
28                sparsematmult.JGFSparseMatmultBench"
29         method="JGFinitialise" descriptor="()V"/>
30    </jPACE:proc>
31
```

Listing 8.4: The original 'JGFinitialise' code that initialises the 'p_datasizes_nz' variable (top), and its associated jPCL characterisation (bottom).

A similar process was performed in order to develop the 'smm-kernel.tran' transaction characterisation of the benchmark's 'JGFkernel' method. This transaction was initially populated with an empty 'method' declaration that included a reference to this 'JGFkernel' method, as well as three 'variable' declarations for 'M', 'nz' and 'p_datasizes_nz' that aided the ACT in defining parameter-dependent expressions. After executing the ACT on this transaction, each loop count's expression was automatically defined. The size of the 'MPIAllreduce' statement was not automatically set however, as the length of the array 'y' could not be calculated, so it was instead manually set to the value of the model's parameter 'M'. Furthermore, the characterised 'case' statements were manually modified to 'MPIcase' statements to

capture the computation that occurs on the master processor. The initialisation code for the value of '`p_datasizes_nz`' was finally inserted into the entry '`proc`' declaration in the same manner as that of the '`smm-init.tran`' transaction.

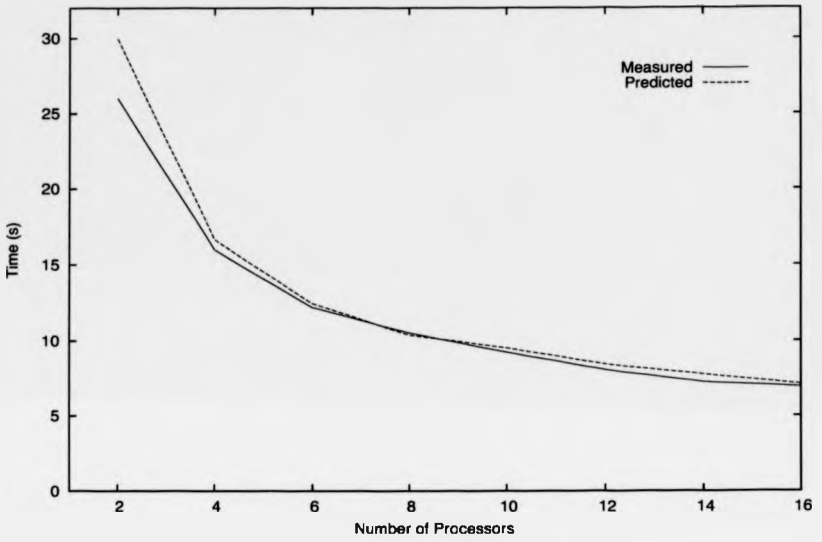
8.1.2 Evaluating the Model

Once the application was characterised, the Hotspot JVM platform implementation was populated with the performance objects that describe the '`mscs`' 16-node cluster's execution environment. A Hotspot JVM virtual machine object was created, which defines the runtime optimisation of the JVM installed on each '`mscs`' workstation as 1500 iterations (the value measured during the benchmarking of bytecode blocks for the Sun Intel Linux JVM v1.4.1_01 as documented in Chapter 6). Sixteen resource objects for each '`mscs`' machine were populated with pre- and post-optimisation benchmark timings for each of the bytecode blocks defined by the ACT while processing each transaction; these benchmark timings were obtained by the automated bytecode block benchmarking tool. Furthermore, three '`methodTiming`' declarations that characterise the response time of the '`java.lang.Math`' and '`java.lang.Random`' API methods where added to these resource objects; these response times were obtained by measuring these method's average execution times over 10 million iterations. A '`mscs`' MPI domain object was created that characterises the MPI communication performance of the Ethernet network, and contains both timings for a '`MPISsend`' to '`MPIRecv`' communication of bytes and a collective '`Allreduce`' communication for a range of processor sizes. The benchmarked timings for bytecode block computation can be found in Appendix C.

Once the platform implementation was populated with these timings, the evaluation engine was used to predict the performance of this benchmark for three different dataset sizes over a varied number of processors. The benchmark was then executed using these parameters over the same range of processors in order to measure the accuracy of the evaluated predictions. The measured and predicted data obtained during

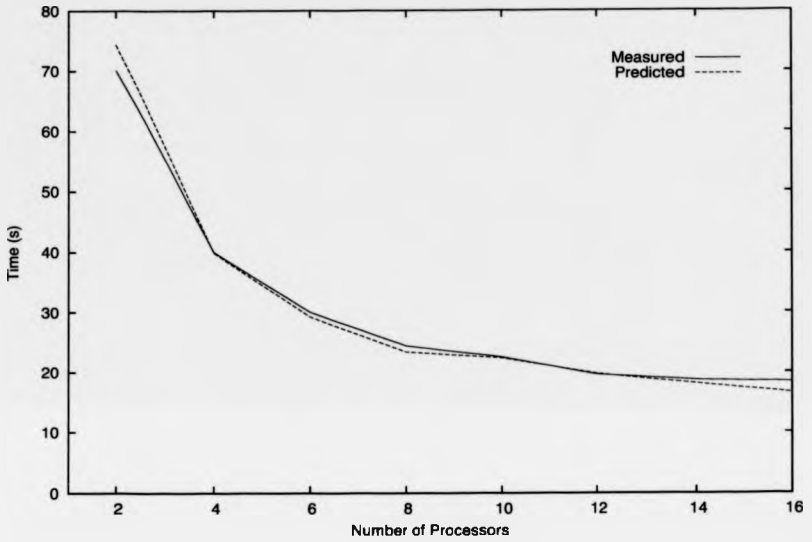
these three experiments is shown in Graphs 8.1, 8.2 and 8.3. The range of predictive inaccuracy achieved from evaluating this model was between 0.35% and 17.63%.

Not included within these results is the evaluated performance's associated confidence. As there were no data-dependent elements within the benchmark's characterisation and all the timings used during evaluation were trusted (including the Java API method calls as the timings measured were for a significantly high number of iterations) this confidence was always evaluated to the maximum value, as defined within the model's application object. The variation of confidence with un-trusted data-dependent characterisations and historical data is documented in the following chapter where the automated refinement of these performance characterisations is performed.



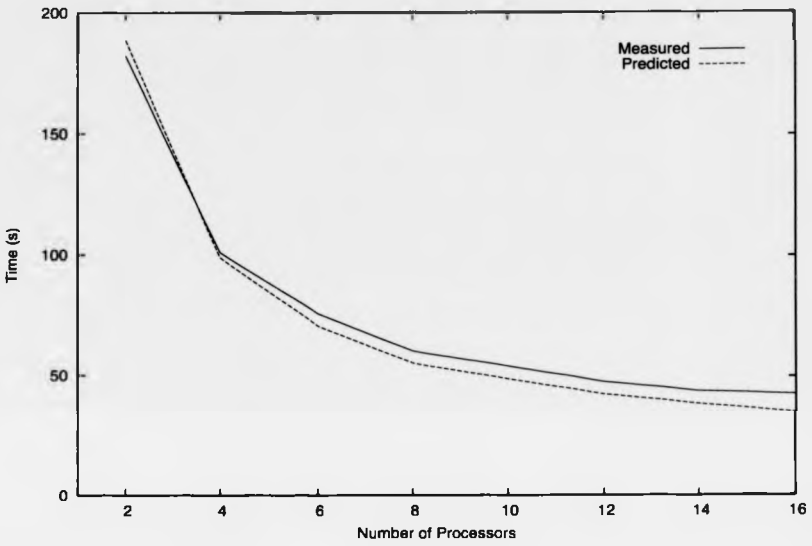
Number of Processors	Measured Execution Time	Evaluated Predicted Time	Predictive Inaccuracy
2	26.010	29.970	15.225%
4	16.000	16.675	4.219%
6	12.207	12.454	2.023%
8	10.524	10.388	1.292%
10	9.238	9.524	3.096%
12	8.064	8.440	4.663%
14	7.254	7.765	7.044%
16	6.943	7.119	2.535%

Graph 8.1: The measured and predicted scalability performance of the Sparse Matrix Multiply benchmark with 'M' and 'N' set to 200000 and 'nz' set to 1000000. All times are in seconds.



Number of Processors	Measured Execution Time	Evaluated Predicted Time	Predictive Inaccuracy
2	70.146	74.369	6.020%
4	39.991	39.850	0.353%
6	30.033	29.245	2.624%
8	24.385	23.368	4.171%
10	22.492	22.351	0.627%
12	19.597	19.732	0.689%
14	18.714	18.140	3.067%
16	18.422	16.620	9.782%

Graph 8.2: The measured and predicted scalability performance of the Sparse Matrix Multiply benchmark with '`M`' and '`N`' set to 500000 and '`nz`' set to 2500000. All times are in seconds.



Number of Processors	Measured Execution Time	Evaluated Predicted Time	Predictive Inaccuracy
2	182.574	188.937	3.485%
4	101.589	99.458	2.098%
6	76.257	71.121	6.735%
8	60.675	55.706	8.190%
10	54.436	49.132	9.744%
12	47.933	42.800	10.709%
14	44.075	38.762	12.054%
16	42.784	35.243	17.626%

Graph 8.3: The measured and predicted scalability performance of the Sparse Matrix Multiply benchmark with 'M' and 'N' set to 800000 and 'nz' set to 6400000. All times are in seconds.

8.2 Fourier Coefficient Analysis

The Fourier Coefficient Analysis JavaGrande benchmark computes the first '`N`' pairs of Fourier coefficients of the function '$f(x) = (x+1)^x$' on the interval 0 to 2. These Fourier coefficients are calculated using a simple trapezoid integration algorithm with 1000 trapezoidal sections. The value of '`N`' is defined within the benchmark as the length of the array '`array_rows`' and is the benchmark's single parameter.

The benchmark is parallelised by splitting '`array_rows`' roughly evenly among the available processors. No communication occurs at the start of the benchmark. Each processor computes the Fourier coefficients assigned to it and populates a local array called '`p_array_rows`' with the results. On finishing, each processor sends the contents of its local array to the master processor, that uses the information to populate the final '`array_rows`' array. This array is then validated before the benchmark concludes.

The performance-critical element of this benchmark is implemented in the '`JGF kernel`' method. It is this method that is characterised within the benchmark's jPCL performance model. The full source code of this benchmark is shown in Appendix A.

8.2.1 Characterising the Application

In order to quickly develop a performance model, it was decided to characterise the benchmark's '`JGFkernel`' method as a single transaction. Similar application and transaction map performance objects (called '`series.app`' and '`series.tranmap`' respectively) to those developed for the Sparse Matrix Multiply benchmark were created. In order to evaluate the model on the same 16-node '`mscs`' Linux cluster, the application object contains the same sixteen '`platform`' declarations as those defined within the previous benchmark's application object. However, this benchmark only defines one parameter and, to reflect this, the application object contains the single '`parameter`' declaration called '`array_rows`'. The application object's entry

'proc' declaration contains one statement that evaluates the model's transaction map, and a 'link' declaration passes the model parameter's value to this transaction map upon evaluation.

The model's transaction map object simply evaluates the 'JGFkernel' method's transaction characterisation on all the platforms currently being evaluated. The transaction map's entry 'proc' contains one statement that evaluates the object's 'map' declaration, and this 'map' declaration contains a single 'step' declaration that evaluates the transaction. When evaluated, a specified 'link' declaration passes the value of the model's parameter to this transaction. Due to the characterisation of this benchmark as one transaction, its predicted performance obtained from evaluating the model will be equivalent to the evaluated performance of the model's single transaction.

The ACT was again used to automate the characterisation of the benchmark's 'JGFkernel' method. The benchmark's bytecode was parsed and decompiled into three 'method' declarations of type 'characterised' that describe the performance of the benchmark's 'Do', 'TrapezoidIntegrate' and 'thefunction' methods. However, the ACT was less helpful this time in automating the calculation of parameter-dependent expressions within the 'method' characterisations, so many of them had to be instead entered manually. The source code of a specific expression that the ACT could not calculate automatically is shown in Listing 8.5. The loop count of this iterative element of code is dependent upon the value of 'ilow', which is initialised to either 1 or 0 depending on the processor's rank. This conditional initialisation of the variable meant that the ACT could not evaluate its value prior to the loop and therefore could not calculate an expression for the loop's loop count. In order to characterise this behaviour, an 'MPIcase' declaration that sets the correct loop count for each processor was inserted into the characterisation. This modified jPCL characterisation is shown in Listing 8.6.

The benchmark's 'thefunction' method specifies which mathematical expression is used to perform the Fourier integration at any one point during the bench-

```
108
109         if(JGFSeriesBench.rank==0) {
110             ilow = 1;
111         } else {
112             ilow = 0;
113         }
114
115         for (int i = ilow; i < p_array_rows; i++) {
116
117             p_TestArray[0][i] =
118                 TrapezoidIntegrate((double)0.0, (double)2.0, 1000,
119                                    omega * ((double)i +
120                                             (ref_p_array_rows *
121                                              JGFSeriesBench.rank)), 1);
122
123             p_TestArray[1][i] =
124                 TrapezoidIntegrate((double)0.0, (double)2.0, 1000,
125                                    omega * ((double)i +
126                                             (ref_p_array_rows *
127                                              JGFSeriesBench.rank)), 2);
128         }
129
```

Listing 8.5: A portion of the Fourier Coefficient Analysis benchmark's 'Do' method.

```
155
156   <jPACE:MPIcase>
157     <jPACE:probValue platform="0">
158       <jPACE:loop count="${p_array_rows} - 1">
159
160         <jPACE:bytecodeBlock id="Do()V:8"/>
161         <jPACE:callMethod class="uk.ac.warwick.dcs.hpsg.applications.jgf.
162                                  series.SeriesTest"
163                           method="TrapezoidIntegrate" descriptor="(DDIDI)D"/>
164         <jPACE:bytecodeBlock id="Do()V:9"/>
165         <jPACE:callMethod class="uk.ac.warwick.dcs.hpsg.applications.jgf.
166                                  series.SeriesTest"
167                           method="TrapezoidIntegrate" descriptor="(DDIDI)D"/>
168         <jPACE:bytecodeBlock id="Do()V:10"/>
169
170       </jPACE:loop>
171     </jPACE:probValue>
172     <jPACE:probValue platform="1 -- (${nP} - 1)">
173       <jPACE:loop count="${p_array_rows}">
174
175         <jPACE:bytecodeBlock id="Do()V:8"/>
176         <jPACE:callMethod class="uk.ac.warwick.dcs.hpsg.applications.jgf.
177                                  series.SeriesTest"
178                           method="TrapezoidIntegrate" descriptor="(DDIDI)D"/>
179         <jPACE:bytecodeBlock id="Do()V:9"/>
180         <jPACE:callMethod class="uk.ac.warwick.dcs.hpsg.applications.jgf.
181                                  series.SeriesTest"
182                           method="TrapezoidIntegrate" descriptor="(DDIDI)D"/>
183         <jPACE:bytecodeBlock id="Do()V:10"/>
184
185       </jPACE:loop>
186     </jPACE:probValue>
187   </jPACE:MPIcase>
188
```

Listing 8.6: The modified characterisation of the model's 'Do' 'method' declaration.

mark's execution (Listing 8.7). The method contains a '`switch`' statement that returns the correct result that depends on the value of the method's parameter and is invoked 1000 times during each execution of the '`TrapezoidIntegrate`' method. In order to associate conditional probability expressions with each case, the application was studied to determine how may times each of the three functions was executed during the course of the benchmark. From studying the benchmark's '`Do`' method, it was calculated that case '`0`' is executed 1000 times, and cases '`1`' and '`2`' are executed 1000 multiplied by the value of '`p_array_rows`' times, where '`p_array_rows`' is the length of the section of the array for that particular processor. From these values, the probability expressions were calculated and the final characterised jPCL '`case`' statement for this '`method`' declaration is shown in Listing 8.8.

```
235
236     private double thefunction(double x,        // Independent variable.
237                                double omegan,   // Omega * term.
238                                int select)      // Choose type.
239     {
240
241         double returnValue;
242
243         switch(select) {
244
245         case 0: {
246             returnValue = Math.pow(x+(double)1.0,x);
247             break;
248         }
249
250         case 1: {
251             returnValue = Math.pow(x+(double)1.0,x) * Math.cos(omegan*x);
252             break;
253         }
254
255         case 2: {
256             returnValue = Math.pow(x+(double)1.0,x) * Math.sin(omegan*x);
257             break;
258         }
259
260         default: returnValue = 0;
261
262         }
263
264         return returnValue;
265     }
266
```

Listing 8.7: A Fourier Coefficient Analysis benchmark's '`thefunction`' method.

Finally some jPCL code was inserted into the transaction's entry '`proc`' declaration in order to set the value of '`p_array_rows`'. This value is initialised in

```
41
42    <jPACE:method class="uk.ac.warwick.dcs.hpsg.applications.jgf.series.SeriesTest"
43                  method="thefunction" descriptor="(DDI)D" type="characterised">
44
45      <jPACE:bytecodeBlock id="thefunction(DDI)D:1"/>
46
47      <jPACE:case>
48        <jPACE:probValue value="1 / ((2 * ${p_array_rows}) + 1)">
49
50          <jPACE:bytecodeBlock id="thefunction(DDI)D:2"/>
51          <jPACE:callMethod class="java.lang.Math"
52                            method="pow" descriptor="(DD)D"/>
53          <jPACE:bytecodeBlock id="thefunction(DDI)D:3"/>
54
55        </jPACE:probValue>
56        <jPACE:probValue value="${p_array_rows} / ((2 * ${p_array_rows}) + 1)">
57
58          <jPACE:bytecodeBlock id="thefunction(DDI)D:4"/>
59          <jPACE:callMethod class="java.lang.Math"
60                            method="pow" descriptor="(DD)D"/>
61          <jPACE:bytecodeBlock id="thefunction(DDI)D:5"/>
62          <jPACE:callMethod class="java.lang.Math"
63                            method="cos" descriptor="(D)D"/>
64          <jPACE:bytecodeBlock id="thefunction(DDI)D:6"/>
65
66        </jPACE:probValue>
67        <jPACE:probValue value="${p_array_rows} / ((2 * ${p_array_rows}) + 1)">
68
69          <jPACE:bytecodeBlock id="thefunction(DDI)D:7"/>
70          <jPACE:callMethod class="java.lang.Math"
71                            method="pow" descriptor="(DD)D"/>
72          <jPACE:bytecodeBlock id="thefunction(DDI)D:8"/>
73          <jPACE:callMethod class="java.lang.Math"
74                            method="sin" descriptor="(D)D"/>
75          <jPACE:bytecodeBlock id="thefunction(DDI)D:9"/>
76
77        </jPACE:probValue>
78        <jPACE:probValue value="0">
79
80          <jPACE:bytecodeBlock id="thefunction(DDI)D:10"/>
81
82        </jPACE:probValue>
83      </jPACE:case>
84
85      <jPACE:bytecodeBlock id="thefunction(DDI)D:11"/>
86
87    </jPACE:method>
88
```

Listing 8.8: The final characterisation of the '`thefunction`' '`method`' declaration.

the benchmark's '`JGFinitialise`' method, which is not characterised within the benchmark's model as it is not performance-critical and is therefore not automatically characterised by the ACT. In order to automate the creation of this jPCL code that initialises '`p_array_rows`', the ACT would have to be extended in order to search for the values of variables outside of the methods defined initially within the model's transaction object. This would require extra information to be declared within the model in order to instruct the ACT where exactly the variable's value is initialised during

the application's execution. The original source code that sets this value, along with the corresponding jPCL code to appropriately initialise the transaction's '`variable`' declaration. is shown in Listing 8.9.

```
50
51          p_array_rows = (array_rows + nprocess -1) / nprocess;
52          ref_p_array_rows = p_array_rows;
53          rem_p_array_rows = p_array_rows - ((p_array_rows*nprocess) - array_rows);
54          if(rank==(nprocess-1)){
55              if((p_array_rows*(rank+1)) > array_rows) {
56                  p_array_rows = rem_p_array_rows;
57              }
58          }
59
```

```
11
12    <jPACE:proc name="main">
13      <jPACE:setVariable variable="p_array_rows"
14                         value="(${array_rows} + ${nP} - 1) / ${nP}"/>
15
16      <jPACE:if leftExpression="${cP}" condition="EQUALS" rightExpression="${nP} - 1">
17        <jPACE:if leftExpression="${p_array_rows}*${nP}" condition="GREATER_THAN"
18                  rightExpression="${array_rows}">
19          <jPACE:setVariable variable="p_array_rows"
20                             value="${p_array_rows} -
21                                    ((${p_array_rows}*${nP}) - ${array_rows})"/>
22        </jPACE:if>
23      </jPACE:if>
24
25      <jPACE:evaluateMethod class="uk.ac.warwick.dcs.hpsg.applications.jgf.
26                                   series.JGFSeriesBench"
27                            method="JGFkernel" descriptor="()V"/>
28    </jPACE:proc>
29
```

Listing 8.9: The original '`JGFinitialise`' code that initialises the '`p_array_rows`' variable (top), and its associated jPCL characterisation (bottom).

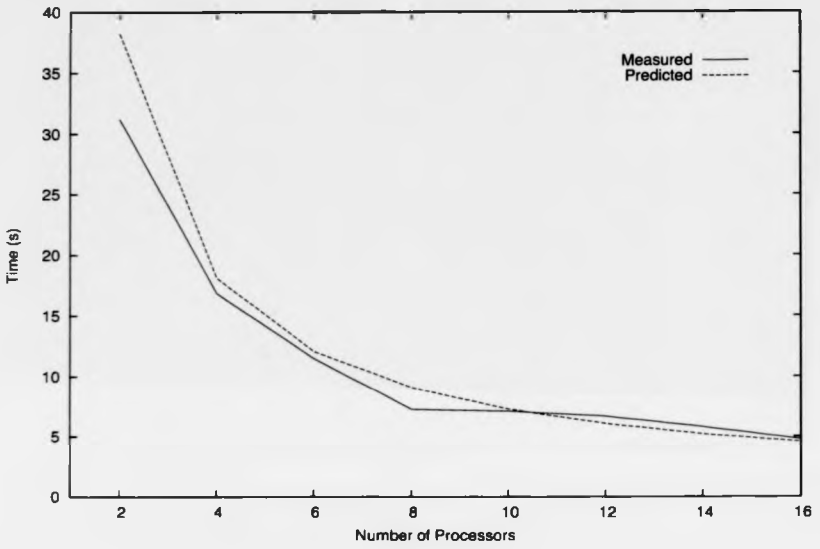
8.2.2 Evaluating the Model

In order to evaluate the benchmark's jPCL model, the transaction's bytecode blocks that were defined by the ACT during characterisation were benchmarked by the automated benchmarking tool. The resultant benchmark timings were used to populate the Hotspot JVM platform implementation's '`mscs`' resource objects, which already contained the block timings for the Sparse Matrix Multiply benchmark. No further communication benchmarking was necessary, as the platform implementation's MPI domain object already contained benchmarked timings for the '`Ssend`' and '`Recv`' MPI communication calls used within this application. In addition, three more '`java.lang.`

`Math`' API calls invoked by the '`thefunction`' method were benchmarked and their average execution time added to each resource object. The platform's bytecode block benchmark benchmark timings obtained using these methods are given in Appendix C.

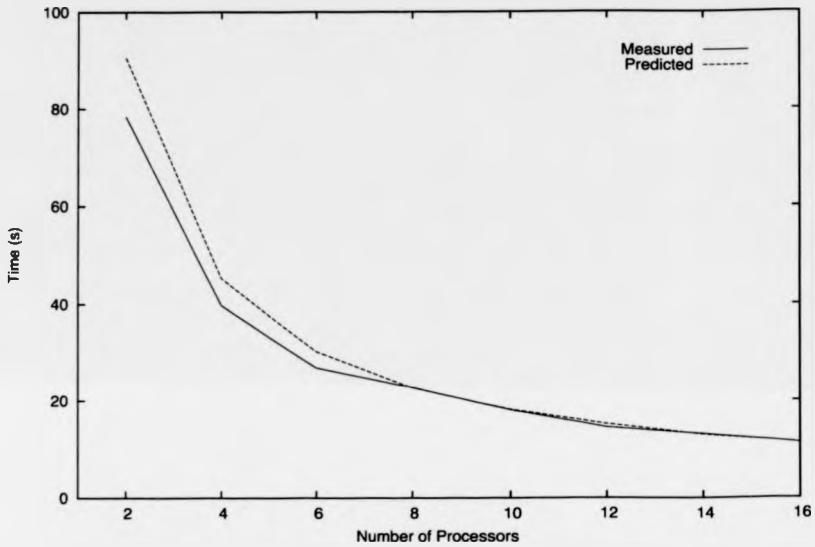
Three different '`array_rows`' values were chosen and the predicted performance of the Fourier Coefficient Analysis benchmark was evaluated over a range of processors for these three dataset sizes. These predicted performances were then compared with the benchmark's actual measured performance with these dataset sizes. The results of these comparisons are shown in Graphs 8.4, 8.5 and 8.6. The range of predictive inaccuracy achieved from evaluating this model was between 0.32% and 25.08%.

Although it is not possible to verify the results, these performance characterisations can be evaluated over a much larger number of processors in order to analyse the scalability performance of the benchmark over larger clusters. Such analysis is useful for both capacity planning (to ensure that purchasing such a system would indeed provide the increase in performance required) and resource allocation services (to determine the number of processors necessary for the application's execution to meet an environmental or user-driven constraint). Two graphs illustrating this benchmark's predicted scalability for a 128-node implementation of the '`mscs`' cluster are shown in Graphs 8.7 and 8.8. These results are made under the assumption that each workstation is within the same MPI domain as the actual 16-node '`mscs`' cluster.



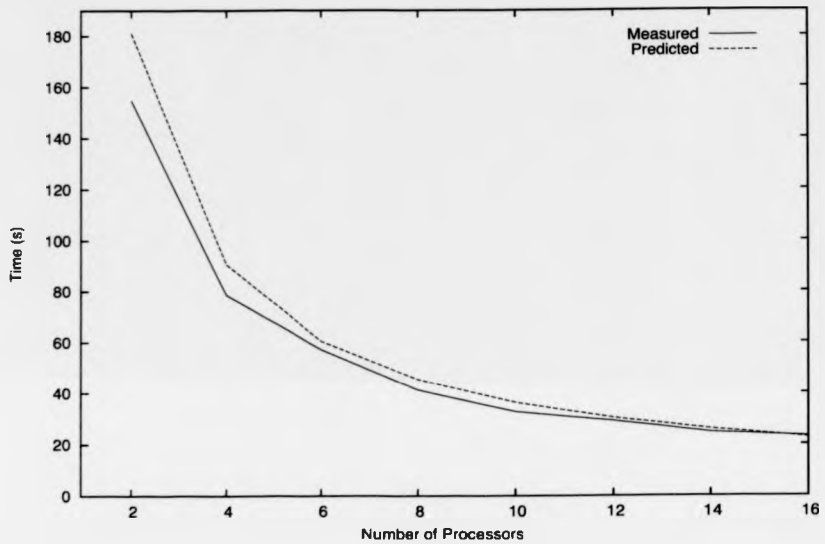
Number of Processors	Measured Execution Time	Evaluated Predicted Time	Predictive Inaccuracy
2	31.221	38.208	22.379%
4	16.851	18.122	7.543%
6	11.507	12.094	5.101%
8	7.260	9.081	25.083%
10	7.097	7.269	2.424%
12	6.673	6.064	9.126%
14	5.783	5.204	10.012%
16	4.782	4.560	4.642%

Graph 8.4: The measured and predicted scalability performance of the Fourier Coefficient Analysis benchmark with '`array_rows`' set to 20000. All times are in seconds.



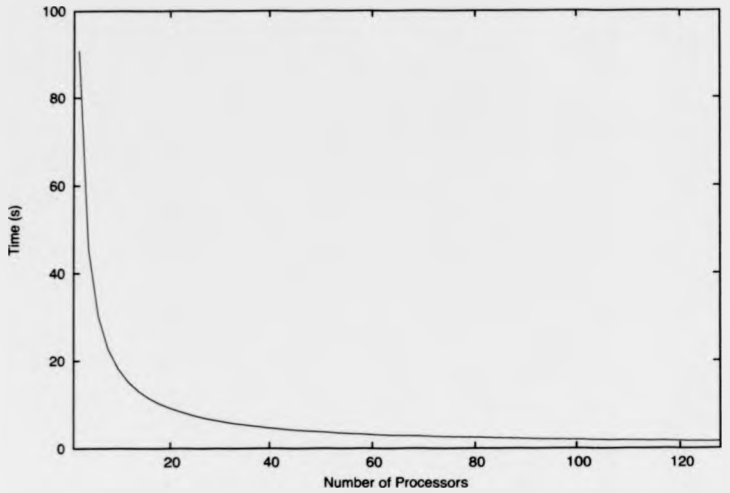
Number of Processors	Measured Execution Time	Evaluated Predicted Time	Predictive Inaccuracy
2	78.362	90.514	15.508%
4	39.753	45.297	13.946%
6	26.898	30.224	12.365%
8	22.761	22.689	0.316%
10	18.074	18.168	0.520%
12	14.487	15.154	4.604%
14	13.137	13.001	1.035%
16	11.344	11.388	0.388%

Graph 8.5: The measured and predicted scalability performance of the Fourier Coefficient Analysis benchmark with '`array_rows`' set to 50000. All times are in seconds.

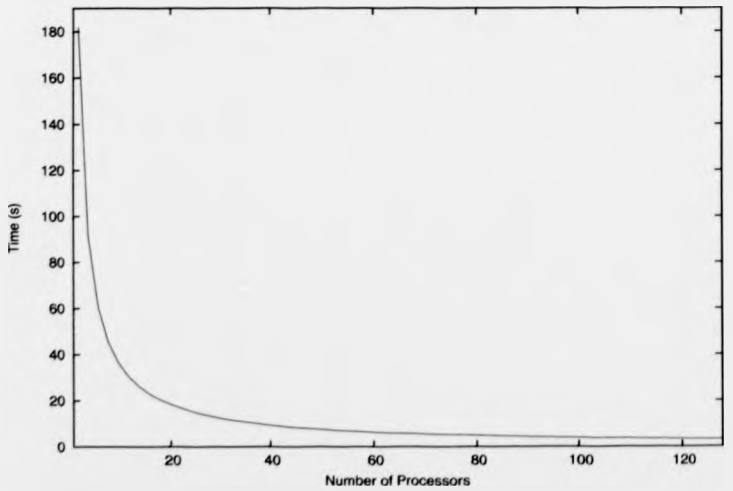


Number of Processors	Measured Execution Time	Evaluated Predicted Time	Predictive Inaccuracy
2	154.699	181.023	17.016%
4	78.585	90.588	15.274%
6	57.150	60.443	5.762%
8	41.212	45.371	10.092%
10	32.689	36.327	11.129%
12	29.108	30.298	4.088%
14	24.706	25.992	5.205%
16	23.185	22.763	1.820%

Graph 8.6: The measured and predicted scalability performance of the Fourier Coefficient Analysis benchmark with '`array_rows`' set to 100000. All times are in seconds.



Graph 8.7: The predicted scalability performance of the Fourier Coefficient Analysis benchmark with '`array_rows`' set to 50000 for a 128-node '`mscs`' cluster. All times are in seconds.



Graph 8.8: The predicted scalability performance of the Fourier Coefficient Analysis benchmark with '`array_rows`' set to 100000 for a 128-node '`mscs`' cluster. All times are in seconds.

8.3 IDEA Encryption

The International Data Encryption Algorithm (IDEA) JavaGrande Benchmark is an MPI implementation of the algorithm, originally developed at ETH Zurich in Switzerland by Xuejia Lai, that was presented by Bruce Schneier in [Schneier96]. The algorithm uses 128-bit keys and is considered very secure among the encryption community. The benchmark initially creates a random 128-bit user key and, from this key, creates 16-bit encryption and decryption sub keys. A byte array of length '`array_rows`' is then populated with random information, encrypted, decrypted and verified. The variable '`array_rows`' that defines the length of this array is a parameter to the benchmark.

The benchmark is parallelised by splitting the array into roughly equal sections. Each section is sent to its associated processor, which encrypts the section, decrypts it, and finally sends the result back to the master processor. Each processor contains a copy of the user, encryption and decryption keys; just the array being encrypted is unique among the processors.

The performance-critical element of this benchmark is implemented in the '`JGF kernel`' method. It is this method that is characterised within the benchmark's jPCL performance model. The full source code of this benchmark is shown in Appendix A.

8.3.1 Characterising the Application

While developing the performance model, it was decided to characterise the IDEA benchmark as three small, sequential transactions, with a detailed transaction map (called '`crypt.tranmap`') that specifically describes each element of MPI communication within the benchmark. This implementation serves to illustrate the detail at which a distributed application can be characterised within jPCL, since the two previous characterisations have been developed with a focus on more rapid evaluation.

The three transactions characterise the initialisation of the '`JGFkernel`' method

prior to communication, the encryption itself and the concluding computation prior to the benchmark's final communication, and are named '`crypt-init.tran`', '`crypt-kernel.tran`' and '`crypt-finalise.tran`' respectively. The complete benchmark's jPCL performance model is provided in Appendix B.

The model's application object is almost identical to that of the Fourier Coefficient Analysis benchmark's model. The 16-node '`mscs`' cluster was defined with the same sixteen '`platform`' declarations as the previous two benchmark characterisations. One '`parameter`' declaration called '`array_rows`' was defined that allows the length of the array encrypted to be modified prior to evaluation. The application's entry '`proc`' declaration contains one statement that evaluates the '`crypt.tranmap`' transaction map; a '`link`' declaration passes the value of '`array_rows`' to this transaction map at the start of its evaluation.

The performance of the benchmark is completely dependent upon the size of each processor's section of the array being encrypted. This size is named '`p_array_rows`' within the benchmark's source code, and its value is set at the beginning of the benchmark by the '`JGFinitialise`' method (as shown in Listing 8.10). This value is the same on all processors apart from the last, whose value may be smaller depending on the size chosen for the complete '`array_rows`' array. Within the transaction map, two '`variable`' declarations are defined whose values are calculated as implemented within the benchmark: '`p_array_rows_general`' is set to the value of '`p_array_rows`' for all processors except the last, and '`p_array_rows_last`' contains the last processor's value. These variables are initialised in the transaction map's entry '`proc`' declaration before the '`map`' declaration is evaluated (see Listing 8.11). The transaction map's '`p_array_rows`' value is then set with the appropriate value prior to evaluating a transaction.

The transaction map's '`map`' declaration (see Listing 8.12) is very similar in structure to the benchmark's '`Do`' method. First of all, the '`crypt-init.tran`' transaction is evaluated on the platform 0 in order to characterise the initialisation

```
85
86      public void JGFinitialise(){
87          array_rows = datasizes[size];
88
89          p_array_rows = (((array_rows / 8) + nprocess -1) / nprocess)*8;
90          ref_p_array_rows = p_array_rows;
91          rem_p_array_rows = p_array_rows - ((p_array_rows*nprocess) - array_rows);
92          if(rank==(nprocess-1)){
93              if((p_array_rows*(rank+1)) > array_rows) {
94                  p_array_rows = rem_p_array_rows;
95              }
96          }
97
98          buildTestData();
99      }
100
```

Listing 8.10: The benchmark's source code that initialises each processor's value of '`p_array_rows`'.

```
15
16    <jPACE:proc name="main">
17      <jPACE:setVariable variable="p_array_rows_general"
18                         value="(((${array_rows} / 8) + ${nP} - 1) / ${nP}) * 8"/>
19      <jPACE:setVariable variable="p_array_rows_last"
20                         value="${p_array_rows_general}"/>
21
22      <jPACE:if leftExpression="${p_array_rows_general}*${nP}" condition="GREATER_THAN"
23                rightExpression="${array_rows}">
24        <jPACE:setVariable variable="p_array_rows_last"
25                           value="${p_array_rows_general} -
26                                  ((${p_array_rows_general}*${nP}) - ${array_rows})"/>
27      </jPACE:if>
28
29      <jPACE:evaluateMap map="crypt.map"/>
30    </jPACE:proc>
31
```

Listing 8.11: The model's '`crypt.tranmap`' transaction map's entry '`proc`' declaration.

code at the start of the benchmark's '`Do`' method. Then a number of MPI statements characterise each communication of the array from the master processor to all other processors. A '`for`' loop is used to evaluate a number of '`step`' declarations for each communication, with a final '`step`' used to characterise the final communication (of perhaps a different size) to the last processor.

The encryption/decryption calculation is then characterised. The sequential '`crypt-kernel.tran`' transaction is evaluated twice within a single '`step`', once to characterise the value of '`p_array_rows`' on the last processor and once to characterise this value of all the other processors. This '`step`' declaration will return the per-

```
31
32    <jPACE:map name="crypt.map">
33
34      <jPACE:setVariable variable="p_array_rows" value="${p_array_rows_general}"/>
35      <jPACE:step>
36        <jPACE:evaluateTransaction transaction="crypt-init.tran" platforms="0"/>
37      </jPACE:step>
38
39      <jPACE:for variable="i" startValue="1" endValue="${nP} - 2" increment="${i} + 1">
40        <jPACE:step>
41          <jPACE:MPISsend destAPI="Recv" source="0" dest="${i}"
42                          datatype="MPI.BYTE" size="${p_array_rows_general}"/>
43        </jPACE:step>
44      </jPACE:for>
45      <jPACE:step>
46        <jPACE:MPISsend destAPI="Recv" source="0" dest="${nP} - 1"
47                        datatype="MPI.BYTE" size="${p_array_rows_last}"/>
48      </jPACE:step>
49
50      <jPACE:for variable="i" startValue="1" endValue="2" increment="${i} + 1">
51        <jPACE:step>
52          <jPACE:setVariable variable="p_array_rows" value="${p_array_rows_general}"/>
53          <jPACE:evaluateTransaction transaction="crypt-encrypt.tran"
54                                     platforms="0 -- (${nP} - 2)"/>
55
56          <jPACE:setVariable variable="p_array_rows" value="${p_array_rows_last}"/>
57          <jPACE:evaluateTransaction transaction="crypt-encrypt.tran"
58                                     platforms="${nP} - 1"/>
59        </jPACE:step>
60      </jPACE:for>
61
62      <jPACE:setVariable variable="p_array_rows" value="${p_array_rows_general}"/>
63      <jPACE:step>
64        <jPACE:evaluateTransaction transaction="crypt-finalise.tran" platforms="0"/>
65      </jPACE:step>
66
67      <jPACE:for variable="i" startValue="1" endValue="${nP} - 2" increment="${i} + 1">
68        <jPACE:step>
69          <jPACE:MPIRecv sourceAPI="Ssend" source="${i}" dest="0"
70                         datatype="MPI.BYTE" size="${p_array_rows_general}"/>
71        </jPACE:step>
72      </jPACE:for>
73
74      <jPACE:step>
75        <jPACE:MPIRecv sourceAPI="Ssend" source="${nP} - 1" dest="0"
76                       datatype="MPI.BYTE" size="${p_array_rows_last}"/>
77      </jPACE:step>
78
79    </jPACE:map>
80
```

Listing 8.12: The model's '`crypt.tranmap`' transaction map's '`map`' declaration.

formance of the transaction with the largest evaluated response time; the '`setVariable`' statements return a response time of 0, as defined within the jPCL specification, and will therefore not affect the result. The '`for`' loop is used to evaluate this '`step`' twice; once for the encryption and once for the decryption.

The '`map`' concludes by evaluating the '`crypt-finalise.tran`' transaction on platform 0, in order to characterise the final computation of the benchmark's

'Do' method, and characterising the benchmark's final communication. This communication is similar in nature to that previously characterised within this 'map', except that the 'MPIRecv' statement replaces the 'MPISsend'. Whether a 'MPISsend' or 'MPIRecv' statement is defined here has no impact on the eventual evaluated performance of the 'map', as both statements are evaluated to the same predicted performance. However, specifying the communication in this way helps to clarify the distributed nature of the benchmark, with the master processor first sending the data out and then, once the encryption has completed, receiving back each processor's section of the array.

Each transaction was created manually by selecting the relevant sections of the benchmark's 'Do' and 'cipher_idea' method characterisations. Both methods were characterised by the ACT into two transaction 'method' declarations, which were then used to populate the model's three transactions as follows:

1. The bytecode that executes on just the master processor prior to the 'Do' method's initial communication was copied into the 'crypt-init.tran' transaction's single 'method' declaration;

2. The bytecode that executes on just the master processor prior to the 'Do' method's final communication was copied into the 'crypt-finalise.tran' transaction's single 'method' declaration;

3. The 'cipher_idea' 'method' declaration created by the ACT was copied into the 'crypt-encrypt.tran' transaction.

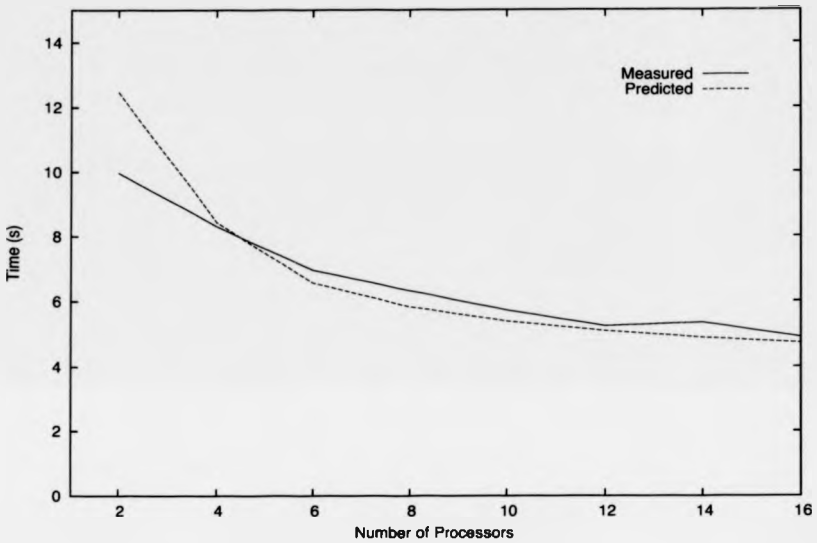
Finally, an entry 'proc' declaration was inserted into each of the above transactions. Each 'proc' declaration contained a statement that evaluates each transaction's respective 'method' declaration.

8.3.2 Evaluating the Model

Once the model was developed, the predicted evaluated time was compared with the measured execution time in the same way as documented for the two previous benchmarks. The Hotspot JVM platform implementation's resource objects were populated with the IDEA Encryption's bytecode block's benchmark timings, as specified by the ACT during the algorithm's automated characterisation.

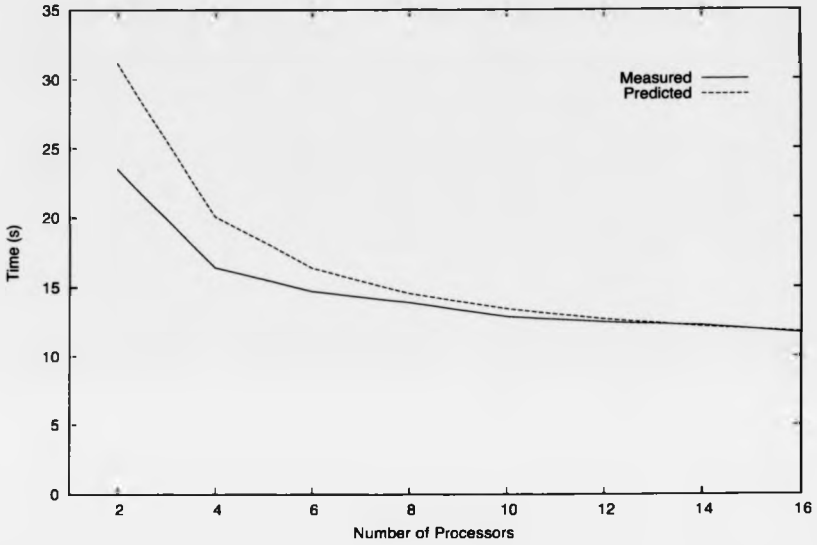
Further MPI domain timings were not required as only the '`MPISsend`' and '`MPIRecv`' communication APIs were characterised, and these had been benchmarked previously. Furthermore, it was not necessary to benchmark any more Java API methods, as none were invoked during the IDEA Encryption's execution. Graphs 8.9, 8.10 and 8.11 contain the results obtained from comparing the model's predicted performance with the benchmark's measured performance. The range of predictive inaccuracy achieved from evaluating this model was between 0.61% and 32.42%. Graphs 8.12 and 8.13 illustrate the predicted scalability analysis of this benchmark for a 128-node implementation of the '`mscs`' cluster.

It can be seen from these results that the majority of the less accurate predictive results were obtained when evaluating this algorithm on a smaller number of processors. This is due to both the very large dataset sizes that were chosen during these experiments and the efficient runtime optimisations of the JVM for repeatedly-executing areas of code. If a small number of processors is chosen, each processor performs the algorithm on a larger portion of the global dataset, and therefore executes the same section of code for a larger number of iterations. While the runtime optimisations of the JVM are taken into account during an evaluation, the further optimisations that can take place for much larger numbers of bytecode iterations are not currently modelled within this implementation. Modelling these different tiers of runtime optimisation is the subject of future work.



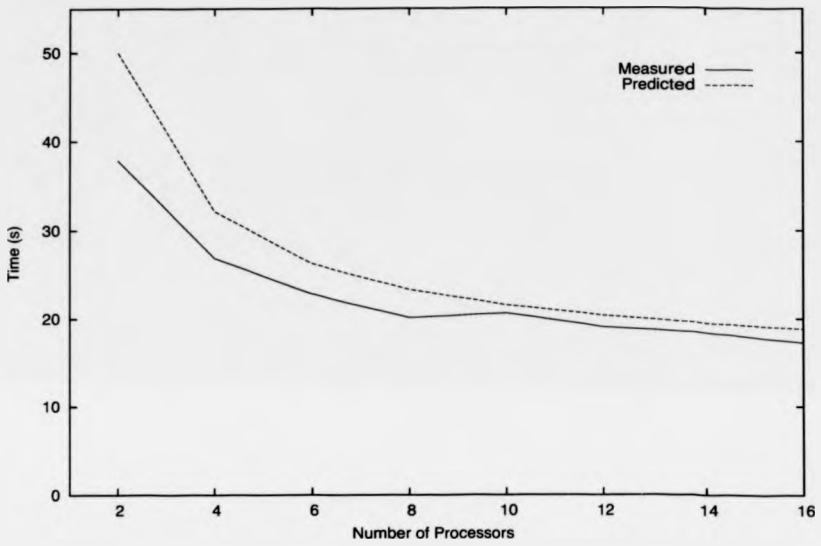
Number of Processors	Measured Execution Time	Evaluated Predicted Time	Predictive Inaccuracy
2	9.973	12.477	25.108%
4	8.305	8.431	1.517%
6	6.957	6.566	5.620%
8	6.325	5.828	7.858%
10	5.724	5.386	5.905%
12	5.238	5.092	2.787%
14	5.345	4.882	8.662%
16	4.898	4.725	3.532%

Graph 8.9: The measured and predicted scalability performance of the IDEA Encryption benchmark with '`array_rows`' set to 20000000. All times are in seconds.



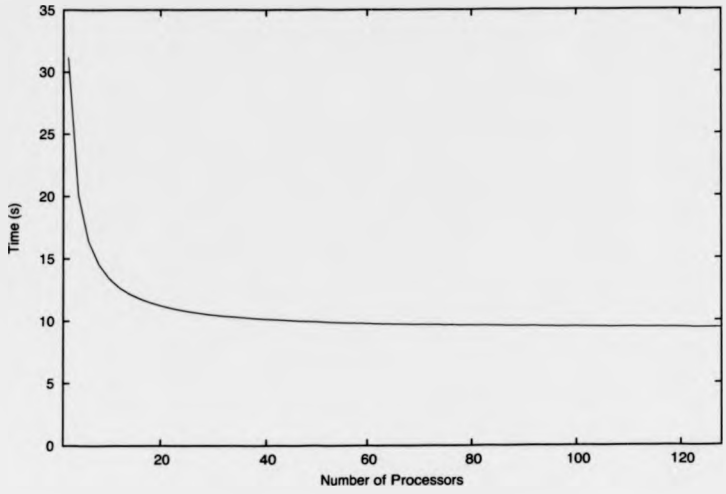
Number of Processors	Measured Execution Time	Evaluated Predicted Time	Predictive Inaccuracy
2	23.549	31.184	32.422%
4	16.433	20.098	22.301%
6	14.727	16.403	11.377%
8	13.919	14.556	4.576%
10	12.895	13.449	4.296%
12	12.509	12.711	1.610%
14	12.268	12.185	0.683%
16	11.719	11.790	0.606%

Graph 8.10: The measured and predicted scalability performance of the IDEA Encryption benchmark with '`array_rows`' set to 50000000. All times are in seconds.

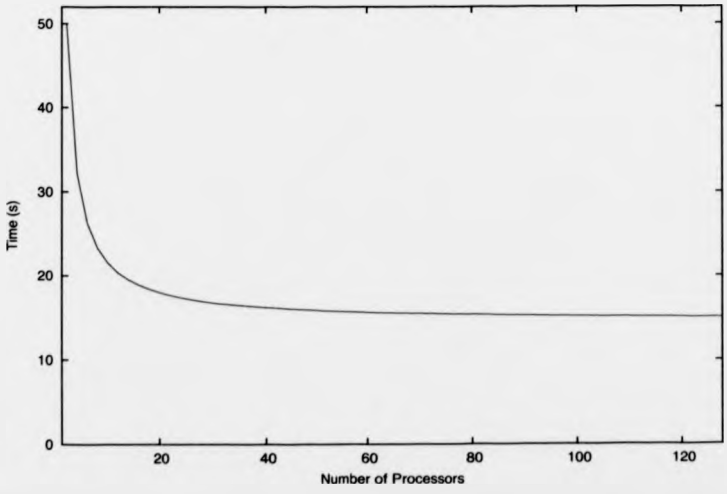


Number of Processors	Measured Execution Time	Evaluated Predicted Time	Predictive Inccuracy
2	37.852	49.981	32.043%
4	26.871	32.152	19.649%
6	22.787	26.240	15.152%
8	20.107	23.285	15.805%
10	20.590	21.512	4.475%
12	19.020	20.331	6.892%
14	18.394	19.487	5.942%
16	17.299	18.855	8.993%

Graph 8.11: The measured and predicted scalability performance of the IDEA Encryption benchmark with '`array_rows`' set to 80000000. All times are in seconds.



Graph 8.12: The predicted scalability performance of the IDEA Encryption benchmark with '`array_rows`' set to 50000000 for a 128-node '`mscs`' cluster. All times are in seconds.



Graph 8.13: The predicted scalability performance of the IDEA Encryption benchmark with '`array_rows`' set to 80000000 for a 128-node '`mscs`' cluster. All times are in seconds.

8.4 Summary

This chapter illustrated the accuracy that can be achieved by using the jPACE framework to predict the performance of distributed scientific Java applications. Three JavaGrande benchmarks were characterised with the jPACE Performance Characterisation Language (jPCL) and evaluated over a range of input dataset sizes and processors. These evaluated response times were then compared with the benchmark's actual measured performance during execution in order to calculate the performance model's accuracy. Each benchmark contained no data-dependent elements of code and could therefore be accurately predicted without any prior execution. From the results obtained, the inaccuracy of the predictive evaluations ranged from 0.31% to 32.4%. Facilitating a predicted performance of this accuracy can be beneficial in efficiently scheduling applications and services within distributed environments.

While this chapter documented the accuracy to which purely parameter-dependent applications can be predicted, evaluating data-dependent applications confidently and with this level of accuracy requires historical data. The following chapter introduces an application monitoring framework that is used to measure this historical data during a characterised application's execution, as well as how this data can be used to automate the refinement of performance models. The predictions of two more JavaGrande benchmarks containing data-dependent elements of computation are documented in order to illustrate this refinement.

Chapter 9

A Monitoring Framework for Automated Predictive Refinement

In the previous chapter, the jPACE framework was used to characterise and evaluate a number of MPI-based Java applications. On comparing the predicted performance with the actual measured performance of these applications, it was documented that a good level of predictive accuracy can be achieved. However, the three applications chosen were completely parameter-dependent, which enables the development of an accurate performance characterisation prior to their execution. If an application contains any data-dependent performance-critical elements, historical data is necessary for a more confident prediction.

This chapter introduces the concept of a monitoring framework, based on the Application Response Measurement (ARM) Java standard, for the automated refinement of jPCL performance characterisations. Any elements of a performance model marked as data-dependent, or any execution timings within a platform implementation that are not trusted, are monitored during the application's execution in order to facilitate more accurate predictions. Characterisation refinement occurs automatically at the beginning and end of a performance model's evaluation; data-dependent expressions are refined at the start and the platform implementation's resource objects at the end.

It is essential to minimise the overhead incurred while monitoring the application's performance during execution. While an implementation of a low-overhead, periodic instrumentation technique is discussed within this chapter, the more elements of the application that are monitored during execution, the greater the overhead. It is therefore important when modelling highly data-dependent applications to reduce the percentage of the characterisation that cannot be confidently predicted. Chapter 3 documented one such approach, where the data was scanned prior to execution in order to capture the complexities of the video stream and enable the reduction of data-dependent elements within the model. Another approach would be to characterise such an application as a single '`method`' declaration of type '`transaction`', thus greatly reducing the detail of the model that would require refinement.

This chapter includes:

- An overview of the Application Response Measurement standard v3.0 for Java. Details of how the performance of applications is monitored and reported with ARM, along with the implementation of a data repository that is used to store this reported information for future use.
- Details of a low-overhead profiling extension to the ARM standard in order to monitor specific conditional, iterative and method invocation elements of an application during execution.
- Documentation regarding how the Java bytecode parser previously described can be used to automate the instrumentation of the appropriate Java bytecode with ARM initialisation and method calls. This is achieved quickly and efficiently, without the necessity for either a knowledge of the ARM specification or the possession of the original source code.
- The details behind the use of this historical data to refine the jPCL performance model characterisations and platform timings.

Two JavaGrande benchmarks containing data-dependent elements of computation are documented in order to illustrate the jPCL refinement. The performance of each benchmark is characterised and its data-dependent elements are located and specified within the model. These elements of the application are then automatically instrumented with the appropriate ARM calls so that their performance is monitored as required during execution. Subsequent evaluations then use this measured historical data to automatically refine the model. It is shown that the model's evaluated response time becomes more accurate and the confidence associated with it rises accordingly after each refinement.

9.1 Application Response Measurement

The Application Response Measurement (ARM) standard [Johnson00] is a framework for the monitoring of distributed items of work or transactions. These transactions can be distributed, they can execute on different systems, across multiple domains and via different processes and threads, as well as implementing different types of work, including database access, application logic, mathematical kernels and data presentation. ARM was developed to provide a basis for analysing the performance of these transactions in order to answer a number of the following performance-related questions:

- Are transactions succeeding?
- If a transaction fails, what is the cause of the failure?
- What is the response time experienced by the end user?
- Which sub-transactions are taking too long?
- Where are the bottlenecks?
- How many of which transactions are being used?

- How can the application and environment be adapted to make it more robust and better performing?

ARM enables the developer to define a number of transactions within an application, whose performance is then measured during execution. Analysing this information enables some or all of the above questions to be answered. The choice regarding which transactions to monitor and their granularity is made by the developer, and will usually correspond to the areas of the application which are considered performance-critical.

Application Response Measurement is employed by 'ARMing' an application. This process requires the developer to instrument the application with a number of ARM interface library calls. These interface calls define where a transaction begins and ends, and the performance of the application between these points is measured and reported to a performance data repository through a consumer implementation of this ARM interface. Via this interface, 'ARMed' applications can communicate with a number of consumer implementations without any change to the original source instrumentation, which allows a variety of performance information to be measured according to the implementation that is used during execution. This communication between an 'ARMed' application, its consumer interface and the performance data repository is summarised in Figure 9.1.

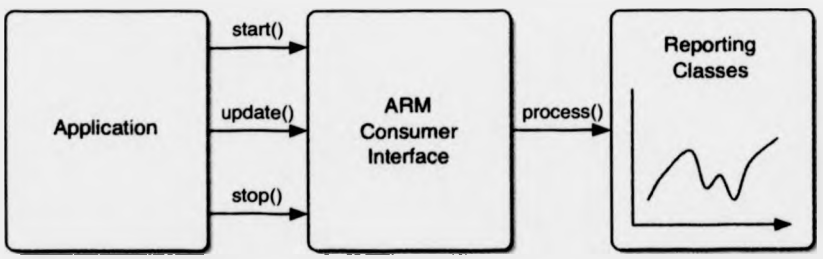


Figure 9.1: An overview of the communication between an application, an ARM consumer interface and the performance data repository classes.

9.1.1 Java 3.0 Binding

The ARM 3.0 specification [OpenGroup01], the final version of which was published by the Open Group in October 2001, provides a Java binding for the response measurement of distributed Java applications. This specification defines a set of Java interfaces, whose method calls are used to define the location and descriptive information of transactions within 'ARMed' applications, and whose implementation by a valid ARM consumer interface is used to measure the performance of such transactions.

Transactions can be simply defined within an application from a number of method calls to the '`ArmTransaction`' interface, the data model for which is presented in Figure 9.2. An instantiation of this interface is obtained from the ARM consumer implementation, and the beginning and end of a transaction is defined by invoking the '`start`' and '`stop`' calls. According to the specification, each transaction records a mandatory set of information, including how long it took for the transaction to successfully complete (the '`response time`'), a time-stamp of when the transaction finished (the '`stop time`') and whether the transaction was successful or not (the '`status`'). Other optional information can be associated with all transactions by using optional definition classes (implementations of the '`ArmUserDefinition`' and '`ArmTranDefinition`' interfaces) and/or a number of optional metrics, which would be included alongside the mandatory metrics defined by the specification.

With the use of correlators, it is possible to create an understanding of how transactions relate within an application. Current and parent correlators can be associated with each transaction, enabling the flow of transactions within an application and across multiple resources to be measured, reported and studied at a later date. This measured relation between transactions is analogous to the characterised relationship as specified within a jPCL transaction map's '`map`' declaration.

Version 3.0 of the ARM specification allows an application to measure performance data itself and report the data asynchronously via the '`ArmTranReport`' interface. This method of transaction performance measurement is known as 'Asyn-

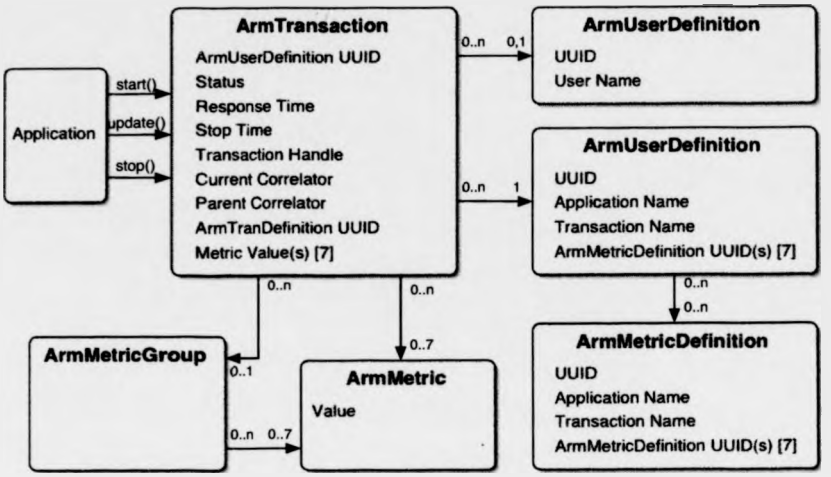


Figure 9.2: The ARM 3.0 Specification Transaction Data Model [OpenGroup01].

chronous Data Reporting'. Instead of the ARM consumer interface implementation measuring the performance of transactions from all '`start`' and '`stop`' calls received, the application measures the performance and populates an implementation of '`ArmTranReport`' itself, when convenient to the application. The application then sends a '`process`' call to the consumer interface, which then reports the data to the repository.

This method of transaction performance measurement is called 'asynchronous' because the performance is measured and reported at different times. The length of time between measurement and reporting of this data is down to the application developer's discretion. For the majority of distributed applications it is possible to use either the asynchronous or synchronous methods for measuring the performance of transactions. However, where it is not feasible to place '`start`' and '`stop`' calls exactly where the transaction is defined, the application must populate the '`ArmTranReport`' instance in order for accurate results to be recorded.

9.1.2 'ARMing' Applications

In certain cases a large number of ARM API calls can be needed in order to define a single transaction and its associated definition data within an application. Figure 9.3 shows a detailed description of the ARM method calls between an application, an implementation of an ARM consumer interface, and a performance data repository for the definition of one transaction, as specified within the Java binding of version 3.0 of the ARM specification. As all ARM invocations reference Java interfaces, instantiations of their implementations have to be obtained from the consumer interface. This is achieved through the use of a number of factories. Instantiations of both definition and transaction object implementations are obtained by calls to the consumer interface's implementation of the '`ArmDefinitionFactory`' and '`ArmTransactionFactory`' respectively.

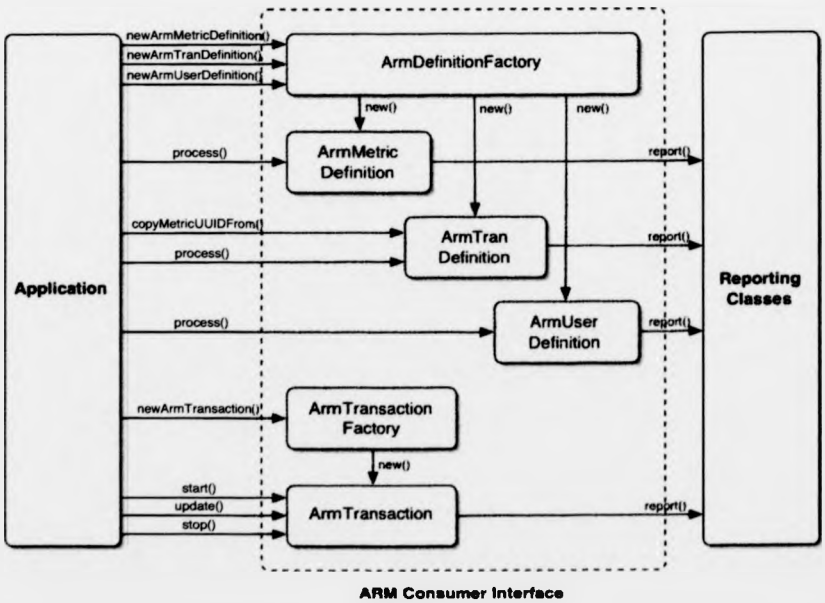


Figure 9.3: The ARM API calls for the description and definition of one transaction between the application, an implementation of an ARM consumer interface, and the reporting classes.

However, an implementation of these factory interfaces must still be obtained. The ARM specification suggests a number of ways to provide an 'ARMed' application with the names of these required factory classes, including: adding a number of system properties to the JVM where the application will be executing; a configuration file that could be passed during execution and could point to the required package names; an abstract '`ArmFactory`' class that could return instantiations of the implemented factories from the correct consumer interface. An example '`ArmFactory`' class is shown in Listing 9.1.

```
1
2   package org.opengroup.arm3.application;
3   import uk.ac.warwick.dcs.hpsg.pace.arm.implclient.*;
4
5   public abstract class ArmFactory {
6
7       public static ArmDefinitionFactory createArmDefinitionFactory() {
8           return new PACEDefinitionFactory();
9       }
10
11      public static ArmTransactionFactory createArmTransactionFactory() {
12          return new PACETransactionFactory();
13      }
14
15      public static ArmTranReportFactory createArmTranReportFactory() {
16          return new PACETranReportFactory();
17      }
18
19      public static ArmMetricFactory createArmMetricFactory() {
20          return new PACEMetricFactory();
21      }
22
23  }
24
```

Listing 9.1: An example abstract '`ArmFactory`' class returning an instantiation of factory classes implemented in the '`uk.ac.warwick.dcs.hpsg.pace.arm.implclient`' package.

Figure 9.3 shows that descriptive and transaction performance information is reported at different times to the performance data repository. It is therefore either a function of the repository or a requirement of the developer to associate descriptive information with the correct transaction performance information. This is achieved through the use of Universally Unique Identifiers (UUIDs). A UUID is associated with each set of information, whether it be a descriptive or performance measurement, and can uniquely identify ARM transaction information among different virtual machines

and systems. For example, to associate a processed 'ArmUserDefinition' with a transaction, the 'ArmUserDefinition' UUID variable within the transaction's 'ArmTransaction' object is set to the UUID of the correct 'ArmUserDefinition' object. All UUIDs are defined by the application when objects are obtained from the different factories, enabling them to be matched up in the future to associate, in this example, a user name with the transaction's performance information.

To measure the performance of a transaction, an 'ArmTransaction' object is obtained from the 'ArmTransactionFactory' and the correct UUIDs and correlators are initialised. At the start of the transaction's execution, a 'start' call to this object is used, and a 'stop' is used when the transaction concludes. Optional 'update' calls can also be invoked for long transactions, in order to provide 'heartbeats' to the consumer interface as a means of confirming its continuing execution and ensuring that the interface does not assume the transaction has failed. By default, the status of the transaction is set to 'ARM_GOOD', representing a successful execution. However, the application could set other more descriptive status flags ('ARM_ABORT' and 'ARM_FAILED' being two such examples) if the transaction does not execute successfully. A 'stop' call to the 'ArmTransaction' object results in the measured performance information of that transaction being reported by the ARM consumer interface to the performance data repository. This information can then be accessed, studied and modified as required.

In order to illustrate the use of ARM to monitor a performance-critical element of an application, a simple example is given: a system administrator has decided that they would like to use ARM to measure the performance of a page request transaction to one of their web servers. The administrator locates the area of the web server's source code that deals with page requests, and instruments the source code with the required ARM API calls as shown in Figure 9.3. Definition information describing the transaction is associated using 'Web Server' as the application name, 'Page Request' as the transaction name, and 'Administrator' as the user name. The instrumented source

code, shown in Listing 9.2, is then recompiled and the instrumented server resumes as normal in order to handle page requests.

An example of the information reported to the performance data repository for one execution of this transaction is shown in Tables 9.1, 9.2 and 9.3. The performance of the transaction is shown in Table 9.1, successfully completing with a response time of 5.638 ms. The item of work to which this transaction information is referring to is clarified by the associated descriptive information, which is matched to the transaction information by the UUID values shown.

TranUUID	UserUUID	Status	Response Time (ms)
64E14	72536	ARM_GOOD	5.638

Table 9.1: Transaction measurements.

TranUUID	Appl Name	Transaction Name
64E14	Web Server	Page Reques

Table 9.2: Transaction definitions.

UserUUID	User Name
72536	Administrator

Table 9.3: User definitions.

9.2 Transaction-based Profiling of Java Applications

ARM provides a good basis for the refinement of jPCL characterisations since the monitoring of transaction performance using ARM can be easily mapped to the transactions defined within jPACE performance models. For a given jPCL transaction's '`method`' declaration, the method could be 'ARMed' as in the previous example and the measured performance used to refine the platform's response objects. However, as it stands, the ARM standard does not have the capability to monitor performance at the same level of abstraction that applications can be characterised at in jPCL. While

```
1
2   public class WebServerExample {
3
4       private static final int NUMBER_TRANSACTIONS = 1;
5
6       private static ArmTransaction[] tranPool;
7       private static byte[][] userUUIDPool;
8
9       static {
10
11          /* Initialise a pool of transactions, and descriptive UUIDs */
12          tranPool = new ArmTransaction[NUMBER_TRANSACTIONS];
13          byte[][] tranUUIDPool = new byte[tranPool.length][ArmConstants.UUID_LENGTH];
14          userUUIDPool = new byte[tranPool.length][ArmConstants.UUID_LENGTH];
15
16          /* Initialise the UUIDs */
17          for (int i = 0; i < tranPool.length; i++) {
18              tranUUIDPool[i] = createNewUUID();
19              userUUIDPool[i] = createnewUUID();
20          }
21
22          /* Initialise the transaction's descriptive information */
23          ArmTransactionFactory armTransactionFactory =
24              ArmFactory.createArmTransactionFactory();
25          ArmDefinitionFactory armDefinitionFactory =
26              ArmFactory.createArmDefinitionFactory();
27
28          String applicationName = "Web Server";
29          String transactionName = "Page Request";
30          String userName = "Administrator";
31
32          ArmTranDefinition armTranDef =
33              armDefinitionFactory.newArmTranDefinition(tranUUIDPool[0],
34                                                        transactionName,
35                                                        applicationName);
36          ArmUserDefinition armUserDef =
37              armDefinitionFactory.newArmUserDefinition(userUUIDPool[0], userName);
38
39          /* Report the descriptive information to the ARM repository */
40          armTranDef.process(); armUserDef.process();
41
42          /* Initialise the transaction pool */
43          tranPool[0] = armTransactionFactory.newArmTransaction(tranUUIDPool[0]);
44
45      }
46
47      public void pageRequest(URL url) {
48
49          /* The beginning of the transaction */
50          tranPool[0].startWithUser(userUUID);
51
52          try {
53
54              *** ORIGINAL PAGE REQUEST SOURCE CODE ***
55
56              /* The end of the transaction */
57              tranPool[0].stop(ArmConstants.ARM_GOOD);
58
59              /* Report the transaction as failed if an exception is thrown */
60          } catch (Exception e) { tranPool[0].stop(ArmConstants.ARM_FAILED); }
61
62      }
63
64  }
65
```

Listing 9.2: The instrumented web server source code. The '`pageRequest`' method has been 'ARMed' appropriately in order to measure the request's performance during execution. The '`WebServerExample`' class' '`static`' method is included to initialise the ARM code.

a number of metrics can be assigned to an ARM transaction as optional descriptive information, their functionality is limited. If ARM is to be used to monitor the data-dependent conditions that can be present within characterised applications, an ARM transaction would need to contain more datatypes that could monitor and report conditional/iterative expressions and sub-transaction method response times.

9.2.1 Low-level Monitors

An extension to ARM has been implemented to enable the profiling of performance-critical low-level elements found within Java applications. At any point during execution, ARM transactions can be populated with a number of monitors, each containing performance information regarding a specific element of the application. The number of monitors is not restricted, enabling transactions to be populated with as many monitors as required for a given item of work. The information that these monitors contain can then be used to automate the refinement of jPCL transactions. The application is required to measure and populate the performance information held within each monitor, regardless of whether the ARM transactions are being reported synchronously (with '`start`' and '`stop`' calls) or asynchronously (via an '`ArmTranReport`' instance).

One of the most performance-hindering aspects of Java is the overhead incurred during a method invocation. As instrumenting an application's low-level elements with ARM calls would result in inappropriate performance overheads, it was decided while developing these extensions not to create a number of new interface calls, each of which would have to be implemented by the ARM consumer in order to measure these element's performance. Instead, a single ARM call that allows collections of monitors to be associated with an ARM transaction instantiation was added to the '`ArmTransaction`' interface.

This extension to the ARM standard defines two types of monitors:

1. *Condition Monitors:* contain a 'before' and 'after' count of conditional and iterative statements. Every time the application reaches a given statement the 'before' count is incremented, and every time it enters that statement the 'after' count is incremented. The result of dividing the 'after' count by the 'before' count is either a conditional statement's probability of execution or an iterative statement's loop count.

2. *Method Monitors:* contain a 'total before clock time', a 'total after clock time' and the 'total number of method invocations'. Just before the method is executed, a timestamp is recorded and added to the 'total before clock time'. When the method returns, another timestamp is recorded, and added to the 'total after clock time', as well as incrementing the 'total method invocation count'. The result of the 'total after clock time' minus the 'total before clock time' is the total execution time of that particular method for all invocations. This execution time divided by the 'total number of executions' results in the method's average execution time.

All monitors contain an ID that references their information to a specific jPCL statement.

9.2.2 Periodic Instrumentation Profiling

In order to populate these monitors, it is necessary to accurately measure the performance of specific elements of the application with as little overhead as possible. Instrumentation and sampling, the two most common techniques for profiling applications, are not suitable in this case. Instrumentation involves the insertion of code into the application that monitors the performance of that area of code every time it is executed. While this is 100% accurate, it results in significant performance overheads. Sampling involves studying the current state of the application's execution at specific time intervals; while this incurs little overhead, this cannot accurately measure the performance

of specific elements of an application.

In order to accurately populate an ARM transaction's monitors with accurate profiling information, while keeping the overhead incurred to a minimum, a periodic instrumentation technique based on [Arnold01] was developed. Performance-critical elements of the application are monitored by instrumenting the application as required. The instrumented code either populates the transaction's monitor with timestamps or increments the monitor's count variables. However, a separate thread executes in the background to periodically enable and disable the instrumentation, dramatically reducing the overhead. While this technique does not result in a complete profile of the application's execution, the information is accurate and can be used as a basis for model refinement.

The use of ARM to monitor the data-dependent elements of the sorting algorithm, introduced in Chapter 5, are described to illustrate this technique. The condition statement (located in the algorithm's '`sort`' method) is used to determine whether two elements of the array need to be reordered, and is therefore data-dependent. Listing 9.3 contains the instrumented source code of the 'ARMed' sort algorithm.

The instrumented code that populates the '`if`' statement's condition monitor is found at lines 102-105 and 109-113. The condition monitor is only populated if the static flag '`conditionFlag0`' is true. This flag is resident within the '`conditionSampler`' class, an instance of a thread that executes in the background during the application's execution and periodically sets all the condition monitor flags to '`true`'. This thread is initialised and invoked in the class's '`static`' method, which is used to initialise the ARM environment. The '`conditionSampler`' thread implementation is shown in Listing 9.4. The '`conditionFlag0`' is only set to '`true`' if the '`conditionFlag0Change`' flag is also '`true`'. This flag is modified by the application in order to stop the sampling threads from modifying a condition flag during the instrumentation's execution.

The instrumented code that populates the '`swap`' method's method monitor is

```
94
95      public void sort() {
96
97          tranPool[0].startWithUser(userUUIDPool[0]);
98
99          for (int i = 0; i < a.length - 1; i++) {
100             for (int j = a.length - 2; j >= i; j--) {
101
102                 conditionSampler.conditionFlag0Change = false;
103                 if (conditionSampler.conditionFlag0) {
104                     conditionMonitors[0].beforeCount++;
105                 }
106
107                 if (a[j] > a[j+1]) {
108
109                     if (conditionSampler.conditionFlag0) {
110                         conditionMonitors[0].afterCount++;
111                         conditionSampler.conditionFlag0 = false;
112                     }
113                     conditionSampler.conditionFlag0Change = true;
114
115                     methodSampler.methodFlag0Change = false;
116                     if (methodSampler.methodFlag0) {
117                         methodMonitors[0].noExecutions++;
118                         methodMonitors[0].totalBeforeTime += NanoTimer.getClockTime();
119                     }
120
121                     swap(j, (j + 1));
122
123                     if (methodSampler.methodFlag0) {
124                         methodMonitors[0].totalAfterTime += NanoTimer.getClockTime();
125                         methodSampler.methodFlag0 = false;
126                     }
127                     methodSampler.methodFlag0Change = true;
128                 }
129             }
130         }
131
132         tranPool[0].stop(ArmConstants.ARM_GOOD);
133
134     }
135
```

Listing 9.3: The 'ARMed' sorting algorithm that implements a periodic instrumentation profiling technique in order to populate the ARM transaction's condition monitor and method monitor objects respectively.

found at lines 115-119 and 123-127. This code is similar to the condition monitor's instrumentation, except that it populates a method monitor instance with timestamps before and after the method's invocation. The instrumentation is only executed if the '`methodSampler`' thread's '`methodFlag0`' flag is true at that point in the execution. Listing 9.5 contains the thread implementation of '`methodSampler`'.

It should be noted that the two sampling thread implementations wait for different amounts of time before enabling their respective instrumentations. The condition monitor instrumentation takes considerably less time to execute than the method mon-

```
1
2   package uk.ac.warwick.dcs.hpsg.pace.arm.profiling;
3
4   public class ConditionSampler implements Runnable {
5
6       private final long conditionSampleInterval = 1;
7
8       public boolean conditionFlag0 = false;
9       public boolean conditionFlag0Change = true;
10
11      public ConditionSampler() {}
12
13      public void run() {
14
15          while (true) {
16
17              try {
18
19                  Thread.sleep(conditionSampleInterval);
20
21                  if (conditionFlag0Change)
22                      conditionFlag0 = true;
23
24              } catch (InterruptedException ie) {}
25
26          }
27
28      }
29
30  }
31
```

Listing 9.4: The condition monitor's thread implementation.

itor instrumentation, as the latter involves a native call to obtain a nanosecond timestamp from the system. To compensate, the method sampler thread enables the method monitor's instrumentation less frequently, further reducing the overhead of populating an ARM transaction's monitors.

An ARM consumer interface and a remote data repository server have been implemented in order to measure and store the data produced from 'ARMed' applications. An overview of this implementation is shown in Figure 9.4. The ARM consumer interface is invoked within the same virtual machine as the application and measures the response time of transactions as defined by the '`start`' and '`stop`' interface methods. Information is processed to a remote data repository service via RMI, which stores this data until it is removed. A number of clients that allow data to be accessed, modified and removed from this repository have also been implemented: a graphical client to visualise the data that is currently stored and a data client that allows remote

```
1
2  package uk.ac.warwick.dcs.hpsg.pace.arm.profiling;
3
4  public class MethodSampler implements Runnable {
5
6      private final long methodSampleInterval = 1000;
7
8      public boolean methodFlag0 = false;
9      public boolean methodFlag0Change = true;
10
11     public MethodSampler() {}
12
13     public void run() {
14
15         while (true) {
16
17             try {
18
19                 Thread.sleep(methodSampleInterval);
20
21                 if (methodFlag0Change)
22                     methodFlag0 = true;
23
24             } catch (InterruptedException ie) {}
25
26         }
27
28     }
29
30 }
31
```

Listing 9.5: The method monitor's thread implementation.

applications to access the repository.

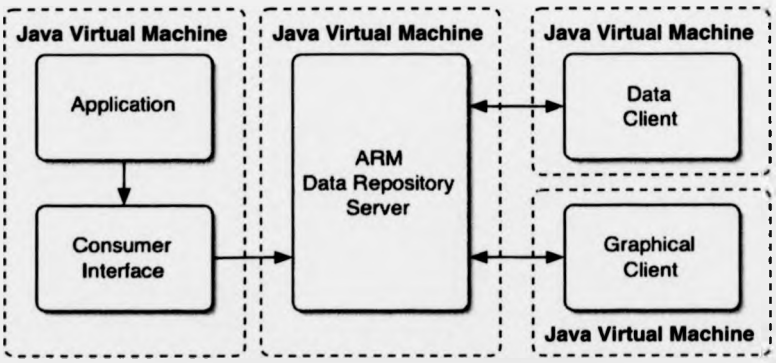


Figure 9.4: An RMI-based ARM interface implementation.

Table 9.4 shows the execution time of the sort algorithm together with the information obtained from the ARM transaction monitors when populated with both the

periodic instrumentation and full instrumentation techniques. An un-'ARMed' algorithm's execution is also shown. Sampling is not shown here as this technique cannot be used to measure specific elements of the application.

Profiling Technique	Response Time (s)	Performance Overhead	'if' Before/ After Count	'if' Probability of Execution	'swap' Iterations	'Average 'swap' Response Time (ns)
None	16.5	N/A	Unknown	Unknown	Unknown	Unknown
Periodic Instrumentation	25.1	52.1%	960/ 485	0.5052%	18	1848.0
Full Instrumentation	1493.8	9000.53%	624373086/ 1249975000	0.4995%	624373086	1640.86

Table 9.4: The performance information obtained from the 'ARMed' sorting algorithm for the sorting of a 50000 element array.

While the information obtained from the periodic instrumentation technique is incomplete, it remains accurate and incurs significantly less overhead than full instrumentation. The majority of the overhead incurred when using periodic instrumentation is from checking the sampling thread flags. The example presented here is also a worst-case estimate of the performance overhead, as the areas of the algorithm that are monitored represent 98% of the transaction's execution time. Generally, where monitors are used evenly throughout the application's execution, the overhead has been significantly less: the overhead incurred from monitoring the two JavaGrande applications as documented at the end of the chapter were less than 1%.

9.3 ARM Bytecode Instrumentation

Extending the ARM standard with a set of monitors that can measure the low-level data-dependent elements of applications, as well as the response times of method invocations, facilitates the use of ARM as a basis for automated model refinement. When using ARM however, once the application's transactions have been specified, its source code must be instrumented with the appropriate ARM API calls as described previously. Not only does this require a detailed knowledge of the ARM specification, but it can also be time-consuming, especially if the application contains many transactions,

each requiring periodic instrumentation to populate their monitor set.

In order to overcome this issue, a tool has been developed to automate the instrumentation of Java bytecode with ARM initialisation and consumer interface code as appropriate. Bytecode instrumentation is achieved with the use of the Java bytecode parser previously documented in this thesis. A model's transactions are processed prior to the characterised application's execution and the bytecode of method specified by the transaction's '`evaluateMethod`' statement is 'ARMed'. This is achieved by inserting an '`invokeinterface`' '`start`' opcode at the point in the original method where the characterisation begins and an '`invokeinterface`' '`stop`' opcode at the point where it ends. If a jPCL statement within the transaction is specified as '`data_dependent`', its corresponding location within the method is found and the bytecode for a condition monitor is inserted. Any '`methodTiming`' declarations containing a '`monitorExecution`' flag result in all appropriate method calls being instrumented with method monitor code. Any methods containing data-dependent expressions are also instrumented with method monitor code. As a transaction's '`confidence`' declaration may be calculated from a method's performance data, inserting this monitor enables the evaluated confidence of data-dependent expressions to also refine appropriately.

A number of static variables that are accessed by the ARM instrumentation during the transaction's execution are added to the method's class. These include: an '`ArmTransaction`' array, which contains instantiations of the method's ARM transactions; an array of '`ArmUserDefinition`' UUIDs, to enable the assignment of user descriptive information to each transaction; arrays of condition and method monitors and implementations of the condition and method sampler threads. If it is not already present within the class, a '`static`' method is inserted and instrumented with the required bytecode in order to initialise these variables. Listing 9.6 shows an example '`static`' method's source code and the static variables that are instrumented[1].

[1] while this instrumentation is actually performed at the bytecode level, the source code is presented here for clarity

```
27
28      static {
29
30          tranPool = new ArmTransaction[NUMBER_TRANSACTIONS];
31          conditionMonitors = new ConditionMonitor[NUMBER_CONDITIONS];
32          methodMonitors = new MethodMonitor[NUMBER_METHODS];
33          byte[][] tranUUIDPool = new byte[tranPool.length][ArmConstants.UUID_LENGTH];
34          userUUIDPool = new byte[tranPool.length][ArmConstants.UUID_LENGTH];
35
36          try {
37
38              ReportServerInterface rsi =
39                  (ReportServerInterface) Naming.lookup("//" + ReportServerInterface.
40                                                        ARM_REPORTSERVER_HOSTNAME +
41                                                        "/ReportServer");
42
43              for (int i = 0; i < tranPool.length; i++) {
44                  tranUUIDPool[i] = rsi.getUUID().getArray();
45                  userUUIDPool[i] = rsi.getUUID().getArray();
46              }
47
48          } catch (Exception e) {
49              System.out.println("Problem retrieving UUID from Report Server"); }
50
51          conditionMonitors[0] =
52              new ConditionMonitor("uk.ac.warwick.dcs.hpsg.applications.misc.
53                                                  bubblesort.BubbleSort.sort()V:pV1");
54          methodMonitors[0] =
55              new MethodMonitor("uk.ac.warwick.dcs.hpsg.applications.misc.
56                                                    bubblesort.BubbleSort.swap(II)V");
57
58          ArmTransactionFactory armTransactionFactory =
59              ArmFactory.createArmTransactionFactory();
60          ArmDefinitionFactory armDefinitionFactory =
61              ArmFactory.createArmDefinitionFactory();
62
63          String executionEnvironment = PACETranDefinition.getExecutionEnvironment();
64          String armTranDef1TranName = "uk.ac.warwick.dcs.hpsg.applications.misc.
65                                                        bubblesort.BubbleSort()V";
66          String armUserDef1UserName = "PACE";
67
68          ArmTranDefinition armTranDef1 =
69              armDefinitionFactory.newArmTranDefinition(tranUUIDPool[0],
70                                                        armTranDef1TranName,
71                                                        executionEnvironment);
72          ArmUserDefinition armUserDef1 =
73              armDefinitionFactory.newArmUserDefinition(userUUIDPool[0],
74                                                        armUserDef1UserName);
75          armTranDef1.process(); armUserDef1.process();
76
77          tranPool[0] = armTransactionFactory.newArmTransaction(tranUUIDPool[0]);
78          tranPool[0].setMonitors(conditionMonitors, methodMonitors);
79
80          conditionSampler = new ConditionSampler();
81          methodSampler = new MethodSampler();
82
83          Thread conditionThread = new Thread(conditionSampler, "conditionSampler");
84          Thread methodThread = new Thread(methodSampler, "methodSampler");
85
86          conditionThread.setDaemon(true); methodThread.setDaemon(true);
87          conditionThread.start(); methodThread.start();
88
89      }
90
```

Listing 9.6: The equivalent source code inserted into the method during the automated instrumentation of transactions.

All transactions are associated with '`ArmUserDefinition`' and '`ArmTran Definition`' objects in order to enable the automated refinement tool to associate historical performance data with a specific jPCL transaction. The automated instrumentation defines the transaction's name as the fully qualified class, name and descriptor of the 'ARMed' method, the transaction's application name to a string that contains the platform's hostname, operating system, processor architecture and the version of the Java virtual machine, and the user name to '`jPACE`'. These descriptive objects are assigned UUIDs from a call to the ARM data repository via the RMI '`ReportServerInterface`' interface object, used by all applications in order to ensure all UUIDs remain unique within a distributed environment and among multiple virtual machines. Once these '`ArmTranDefinition`' and '`ArmUserDefinition`' objects have been initialised with this data, they are processed to the ARM data repository.

Each monitor that is created within the transaction is then initialised. Each monitor's ID contains a reference to the statement or declaration within the performance model whose performance is being measured. Once initialised, the two arrays of monitors are then associated with the transaction. When the transaction is processed by the '`stop`' call at the end of its execution, the values held all associated monitors are reported and stored within the repository, along with the transaction's response time, stop time and the other mandatory metrics defined by the ARM specification.

The final statements entered into the '`static`' method start the monitor condition and method sampling threads. Each thread is instrumented with two flags for each of its associated monitors within the transaction. The thread's '`run`' method is then modified so that all of these flags are set appropriately. All monitor flags are initialised to '`true`' such that the first monitor's instrumentation code is always executed, no matter how early in the transaction's execution.

9.4 Automated Characterisation Refinement

jPCL transactions are automatically refined during evaluation. At the beginning of a transaction's evaluation, the evaluation engine searches the ARM data repository in order to see if there is any historical data relating to that transaction for the particular platform. If there is, the data is copied locally using the RMI-based ARM implementation's data client. Every time a '`data_dependent`' statement within the transaction is evaluated, the local copy of the ARM-measured data is searched to see if a monitor that represents the performance of that statement exists. If it does, the information within that monitor is used to update the appropriate value or expression within the model prior to the statement's evaluation. At the end of evaluation, the refined objects within the performance model are written back to disk and the copy of historical data within the ARM data repository is deleted. The result of this modification is that the eventual evaluated performance of the transaction relates more closely to the transaction's actual runtime performance.

Two sequential JavaGrande applications are used to illustrate this automated refinement. Each application was characterised as a single transaction and created automatically using the ACT. Unlike those predicted in the previous chapter, these applications contain expressions that are either impossible or very complicated to characterise prior to execution. These expressions were characterised as data-dependent within the transaction and monitored during execution by ARM after automated instrumentation in order to refine their characterisation. Each application is executed several times in order to illustrate the change in both predictive accuracy and the evaluated confidence associated within each prediction after each execution. The application's source code, its respective performance models and its bytecode benchmark timings are given in Appendices A, B and C respectively.

The automated refinement of distributed applications, such as those documented in the previous chapter, is also possible using this monitoring framework. The perfor-

mance of a transaction is measured on each platform, resulting in multiple instances of historical data being reported to the ARM data repository when the transaction's execution concludes. This data is then used during evaluation to refine the performance characterisation of that particular platform. Communication API benchmark timings are not monitored as they are trusted performance measurements within the jPACE framework.

9.4.1 Gaussian Random Number Generation

The JavaGrande Gaussian Random Number Generator benchmark generates random number pairs in accordance with the equivalent NAS benchmark's specification and takes one parameter, which is the number of random numbers to generate. The benchmark's transaction object contains one '`probValue`' statement, which was defined data-dependent during automated characterisation and initialised to 1. The associated statement from the source code ('`if t <= 1.0`') cannot be predicted prior to execution without characterising the 64-bit linear congruential generator implemented in the benchmark's '`next`' method. Automated refinement was chosen as the preferable alternative.

Table 9.5 illustrates the changes in predictive accuracy and associated confidence that occurred due to the automated refinement of the model over five benchmark executions. Each execution of the benchmark used different input data sizes as specified by 'n'; the value actually used is 2^n. Listing 9.7 contains the transaction's '`confidence`' declaration from the benchmark's performance model, which states that the characterisation is evaluated with a maximum confidence after five executions; the maximum confidence value is set in the model's application object to one. The '`probValue`' statement's expression is continuously refined over these five benchmark executions and the model's accuracy, and associated confidence, improves accordingly. Each benchmark is executed on and predicted for a '`labvista`' Linux workstation containing an Intel Pentium III 800MHz processor and 128MBs RAM.

```
6
7     <jPACE:confidence variable="c">
8       <jPACE:setVariable variable="c" value="${nE} / 5"/>
9     </jPACE:confidence>
10
```

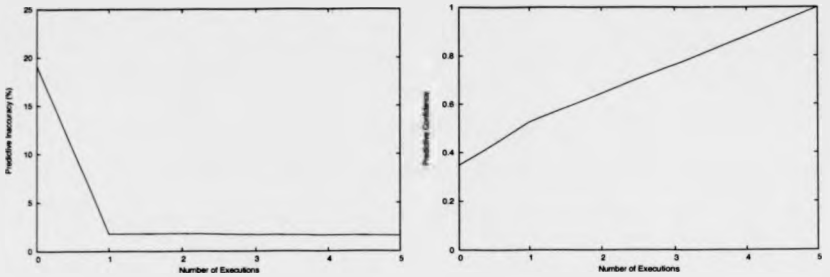
Listing 9.7: The Gaussian Random Number Generator benchmark's '`confidence`' declaration.

Number of Executions	Data Size ('n')	Execution Time (s)	'if' Statement Before/After Count	'if' Statement Refined Value	Evaluated Prediction Response Time (s)	Evaluated Prediction Confidence
0	25	Unknown	Unknown	1	66.031	0.3478
1	25	54.877	15681/12325	0.78598	56.433	0.525
2	28	444.832	127480/100242	0.78616	451.462	0.644
3	22	6.948	1958/1531	0.78475	7.056	0.763
4	24	27.574	7987/6245	0.78403	28.178	0.882
5	25	55.331	15929/12464	0.78372	56.332	1

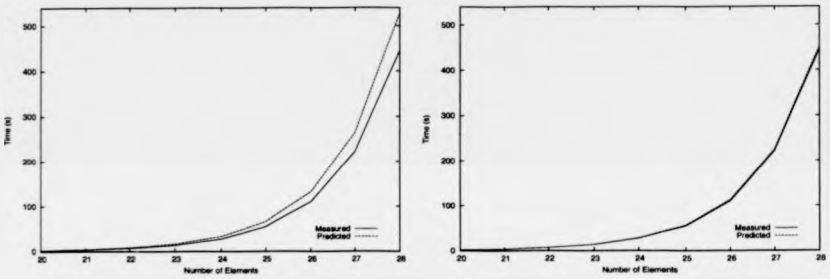
Table 9.5: The automated refinement of the Gaussian Random Number Generator benchmark's performance characterisation.

Table 9.5 shows that after each refinement, the predictive inaccuracy decreases to zero and the associated confidence increases to its maximum value of, in this case, one. Graph 9.1 illustrates this variance in accuracy and confidence over the five executions. This graph illustrates how the predictive inaccuracy tends to dramatically reduce after the first execution, and then continues to decrease gradually from there on. The associated confidence however increases smoothly over each execution. Graph 9.2 compares the predicted and measured response times of this benchmark on the '`labvista`' workstation before and after refinement.

While the execution of this benchmark was achieved on a '`labvista`' workstation, the improvement in predictive accuracy was a result of refining the benchmark's transaction characterisation. Due to the separation of the application and platform characterisations that jPCL employs, this refinement of the benchmark's performance model also inherently increases the accuracy of evaluating this benchmark on other differently-performing resources, without the necessity of actually executing the benchmark of that resource. A comparison of the predicted and measured response times of this benchmark on '`mscs`' and '`budweiser`' workstations, before and after the benchmark's refinement on '`labvista`', is shown in Graphs 9.3 and 9.4 respec-



Graph 9.1: The variance of the average predictive inaccuracy (left) and confidence (right) of the Gaussian Random Number Generator benchmark's model over five executions.

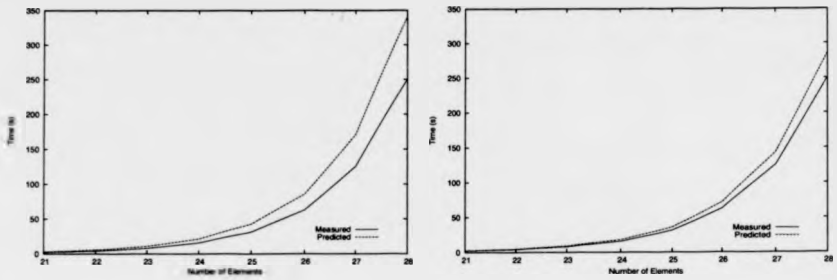


Graph 9.2: A comparison between the measured and predicted response times of the Gaussian Random Number Generator benchmark's model on a '`labvista`' workstation before (left) and after (right) refinement. The number of elements specified here is the benchmark's data size '`n`' as documented previously; the actual data size used for these results is 2^n.

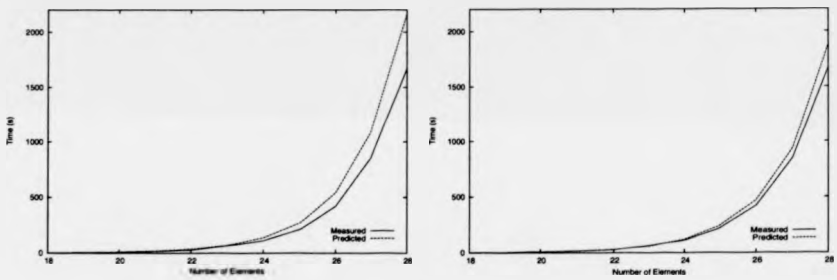
tively. '`mscs`' is a Linux workstation that contains a 2.4GHz Pentium IV processor and 512MBs RAM. '`budweiser`' is a Solaris workstation that contains a UltraSparcII 360MHz processor and 128MBs RAM. These results illustrates the effectiveness of the jPACE framework on heterogeneous architectures.

9.4.2 Fast Fourier Transform

The JavaGrande Fast Fourier Transform benchmark implements a 256-point fast Fourier transform and inverse transform, and takes three parameters (named '`n1`', '`n2`' and '`n3`') that define the size of the three-dimensional array processed. Both the Fourier and inverse transforms are implemented within the benchmark by the '`auxiliary`



Graph 9.3: A comparison between the measured and predicted response times of the Gaussian Random Number Generator benchmark's model on a '`mscs`' workstation before (left) and after (right) refinement. The number of elements specified here is the benchmark's data size '`n`' as documented previously; the actual data size used for these results is 2^n.



Graph 9.4: A comparison between the measured and predicted response times of the Gaussian Random Number Generator benchmark's model on a '`budweiser`' workstation before (left) and after (right) refinement. The number of elements specified here is the benchmark's data size '`n`' as documented previously; the actual data size used for these results is 2^n.

`complex_fourierndʼ` method.

The majority of the iterative loop counts and conditional probabilities within the benchmark's characterisation were expressed in relation to the benchmark's three parameters, either automatically by the ACT or manually afterwards. However, three parameter-dependent statements within the '`auxiliary_complex_fourierndʼ` method (an '`if`' statement, a '`for`' statement and a '`while`' statement) were too complex to characterise prior to execution and were marked as data-dependent for automated refinement. These statements were automatically instrumented with ARM condition monitors in order to measure their loop counts and conditional probability

values during execution.

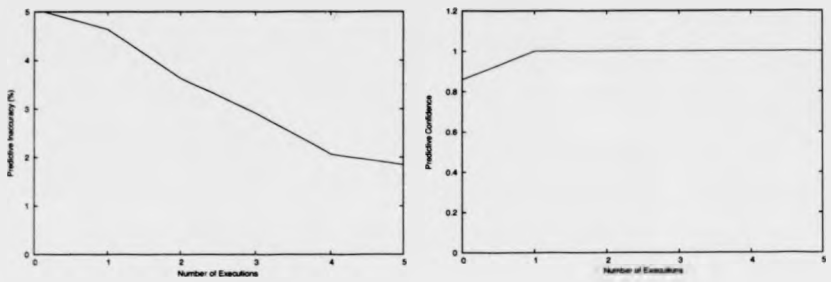
Table 9.6 contains the results of the characterisation's automated refinement over five Fast Fourier Transform benchmark executions of varying dataset sizes. Each iteration of the benchmark is executed on and evaluated for the '`budweiser`' Solaris workstation. In this example, the predicted accuracy of the benchmark's model is high prior to refinement and, while automated refinement does increase the model's accuracy over time, the resulting difference in accuracy is not as significant as in the previous example. However, the initial confidence value prior to execution was also particularly high, suggesting that the evaluated prediction prior to refinement would also be more accurate.

Number of Executions	Data size 'n1'	'n2'	'n3'	Execution Time (s)	Refined 'if'	Refined 'for'	Refined 'while'	Evaluated Prediction Response Time(s)	Confidence
0	128	128	112	Unknown	1	1	1	195.889	0.875
1	128	128	112	188.451	0.4498	4.2583	0.5	195.3345	1
2	32	32	32	2.921	0.4352	4.0854	0.5	3.0066	1
3	64	64	32	14.898	0.4352	4.0854	0.5	16.0018	1
4	128	64	64	55.767	0.4323	3.5587	0.5	57.3744	1
5	64	64	64	24.214	0.4356	3.4438	0.5	24.5356	1

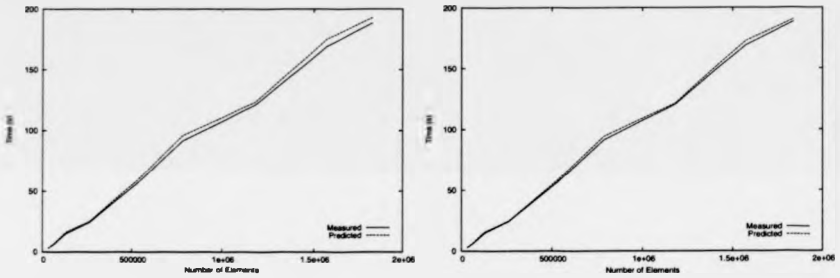
Table 9.6: The automated refinement of the Fast Fourier Transform benchmark's performance characterisation.

Graph 9.5 illustrates the variance of the average predictive inaccuracy obtained during refinement and its associated confidence. It can be seen that the confidence value reaches its maximum value after only one benchmark execution. While the same '`confidence`' declaration as the previous example is used, each execution of the benchmark results in more than one iteration of the '`auxiliary_complex_fourierndd`' method. During the benchmark's first execution, this method was iterated seven times meaning that, after the first refinement, the model's data-dependent expressions are assigned the maximum confidence value. An ARM method monitor is inserted into the benchmark's bytecode during automated instrumentation in order to measure the number of times the method is executed.

Graph 9.6 compares the predicted and measured response times of this benchmark on the '`budweiser`' workstation before and after refinement. As was the case

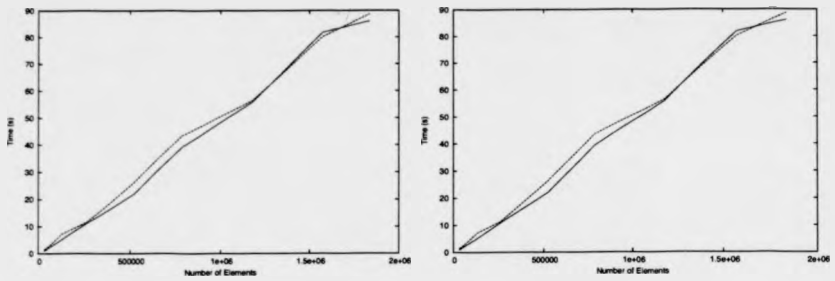


Graph 9.5: The variance of the average predictive accuracy (left) and confidence (right) of the Fast Fourier Transform benchmark's model over five executions.

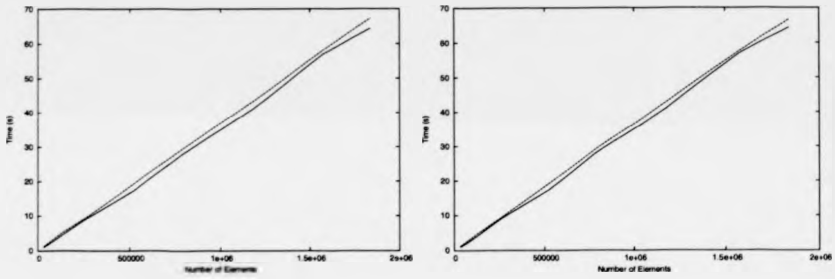


Graph 9.6: A comparison between the measured and predicted response times of the Fast Fourier Transform benchmark's model on a '`budweiser`' workstation before (left) and after (right) refinement. The number of elements specified here is the multiplication of the benchmark's three parameters: '`n1`', '`n2`' and '`n3`'.

for the previous benchmark, this refinement during evaluation is specifically for the benchmark's transaction characterisation, resulting in more accurate predictions on all evaluated platforms. Graphs 9.7 and 9.8 illustrate the benchmark's refinement for the '`labvista`' and '`mscs`' workstations over five executions. However, due to the model's high initial accuracy, the graphs before and after refinement are very similar in appearance as, in this example, the automated refinement only resulted in a very small increase in predictive accuracy over the benchmark's five executions.



Graph 9.7: A comparison between the measured and predicted response times of the Fast Fourier Transform benchmark's model on a '`labvista`' workstation before (left) and after (right) refinement. The number of elements specified here is the multiplication of the benchmark's three parameters: '`n1`', '`n2`' and '`n3`'.



Graph 9.8: A comparison between the measured and predicted response times of the Fast Fourier Transform benchmark's model on a '`mscs`' workstation before (left) and after (right) refinement. The number of elements specified here is the multiplication of the benchmark's three parameters: '`n1`', '`n2`' and '`n3`'.

9.5 Summary

While it is possible to accurately predict the performance of applications prior to any execution (as documented in Chapter 8), historical data is needed in order to evaluate the performance of data-dependent elements of code with a reasonable confidence. This historical data can then be used to refine performance characterisations, thus improving the accuracy of future evaluations. However, a detailed knowledge of both the application and the use of profiling tools is required if the manual refinement of performance characterisations is to be achieved. As shown in Chapter 3, this can be a

time-consuming and labourious task.

This chapter documented a monitoring framework for the automated refinement of performance characterisations. Data-dependent areas of an application's bytecode that cannot be confidently predicted without historical data are located and instrumented with monitors, which measure their performance during execution. This historical data is then processed during evaluation in order to automatically refine the application's performance characterisation, resulting in both more accurate and more confident future evaluations. A number of sequential JavaGrande benchmarks were characterised and refined in order to illustrate this technique.

A number of performance-based middleware services, which use the predictive and monitoring framework discussed in this thesis to increase the efficiency of distributed environments, are currently being developed. These include a Grid resource manager for the efficient scheduling of scientific applications among disparate clusters and a workload management infrastructure for the service routing of transaction-based requests. An overview of these services is documented in the following chapter.

Chapter 10

Performance-based Middleware Services

This chapter introduces two middleware services being developed at the University of Warwick. Each service employs the jPACE prediction and monitoring framework in order to improve the efficiency of modern computing environments. TITAN [Spooner02a, Spooner02b], a Grid scheduler, uses the jPACE framework to predict the performance of submitted applications on the available resources prior to their execution. Performance data is used here to better schedule applications within heterogeneous environments; this is done by using the predicted performance data to reduce makespan and idle time while at the same time ensuring that user-driven deadlines are still maintained. A workload management infrastructure [Bacigalupo03b] is also being developed that uses the jPACE framework to efficiently route requests within an e-business environment. The use of jPACE within these two services is documented in this chapter.

10.1 Performance-based Application Scheduling

TITAN is a Grid resource manager that facilitates the efficient scheduling of applications among distributed, heterogeneous resources. Predictive performance data, ob-

tained from the jPACE framework, is used at both the cluster level and inter-domain level in order to improve resource allocation throughout the distributed computing environment. This two-tier management model is illustrated in Figure 10.1.

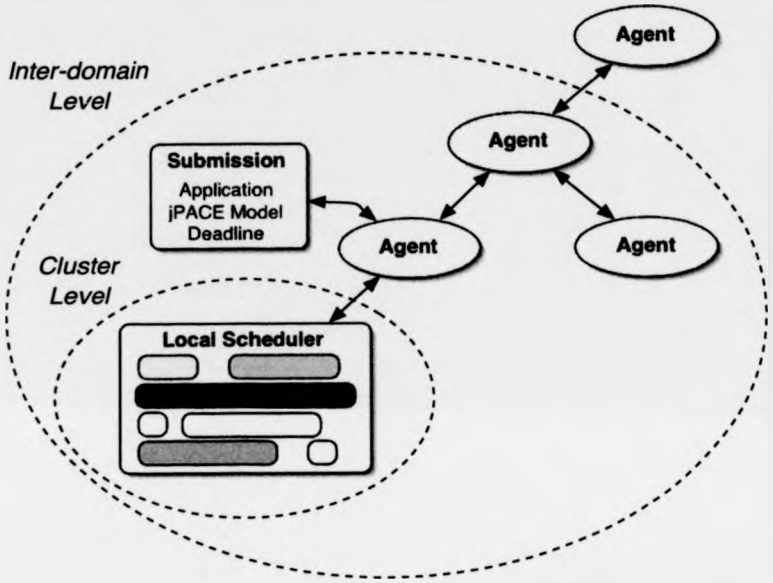


Figure 10.1: The TITAN two-tier resource management model.

A distributed computing environment is modelled within TITAN using an agent hierarchy. Each agent represents a local cluster of resources, which itself can be a multiprocessor machine, a cluster of workstations, and so on. Agents are able to discover neighbouring resources within the environment and advertise their resources to other agents, therefore ensuring a global awareness of the available resources at any point in the distributed system [Cao01c, Cao01b, Cao01a]. Applications are submitted to an agent via a Grid portal and are advertised throughout the hierarchy until a suitable resource is found where it is predicted that the application will best meet its requirements. These requirements are defined by the user during the application's submission to the portal and currently specify the time by which the application's execution must

have finished. The application is then submitted to that resource for execution. It has been shown that balancing the submitted applications among the resources using predicted performance information results in the reduction of makespan (time to complete the submitted tasks) and idle time among the resources, as well as the improvement of the overall system's throughput [Jarvis03a, Spooner03].

TITAN further improves the resource usage by continuously refining the scheduling of applications at the cluster level. An overview of the cluster-level scheduling framework that TITAN employs is shown in Figure 10.2. At this level in the system, TITAN utilises the predicted performance of each application, together with an iterative heuristic algorithm in order to compact the overall time required to execute the set of applications on each local resource. Information regarding the available resources is obtained from a resource monitoring module, which is periodically updated so that it represents the 'current view' of its resources.

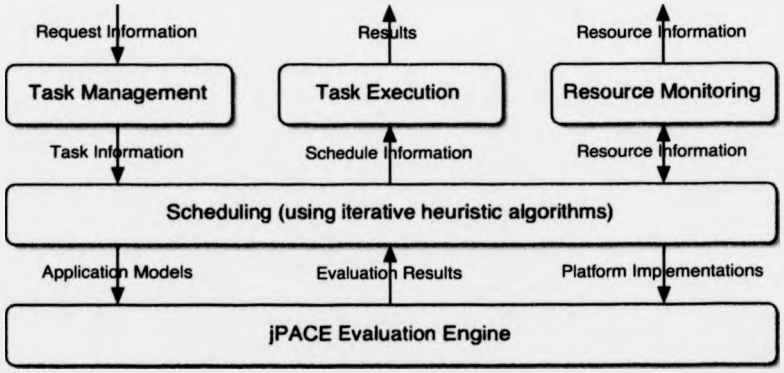


Figure 10.2: Cluster-level task management.

The benefits achieved from applying performance prediction at the cluster level and inter-domain level are illustrated in two experiments that are detailed below. The aim of each experiment is to improve the schedule of JavaGrande benchmarks over a simple 'first-come first-served' schedule by evaluating their performance prior to execution and using this to steer the scheduling. The predicted performance of each

application is obtained by utilising the benchmark's jPACE performance models as documented in Chapters 8 and 9.

10.1.1 Cluster Level Optimisation

In order to illustrate the cluster level optimisation that is employed by TITAN, 32 JavaGrande benchmarks were submitted to a 16-node cluster of UltraSparcII 360MHz Solaris workstations via the cluster's task management console. Figure 10.3 presents the scheduling time-line prior to any heuristic refinement or optimisation. Each row of the figure represents one node of the cluster and each box represents one of the submitted JavaGrande benchmarks. Each application has been placed into the time-line using a 'first-come first-served' approach. It can be seen that, while each application will eventually execute, each node of the cluster remains idle for a considerable portion of the overall scheduling makespan.

Figure 10.4 illustrates the scheduling time-line after 2000 heuristic iterations. By evaluating the performance models associated with each benchmark, the execution time of each application, as well as how each application scales among the nodes within the cluster, is analysed in order to continuously refine the time-line. It can be seen that the sequential benchmarks were scheduled for execution on nodes '`n01`' to '`n08`', while the parallel tasks are to execute on the remaining nodes. Processor idle time and the overall makespan have been greatly reduced; the execution time, for example, is reduced from two hours to forty minutes.

10.1.2 Inter-domain Level Optimisation

In order to illustrate inter-domain level optimisation, 1000 JavaGrande benchmarks are submitted to a distributed computing environment consisting of 16 heterogeneous clusters. The agent hierarchy that represents this environment is shown in Figure 10.5. Agents '`A1`' through '`A16`' represents one of three differently-performing architec-

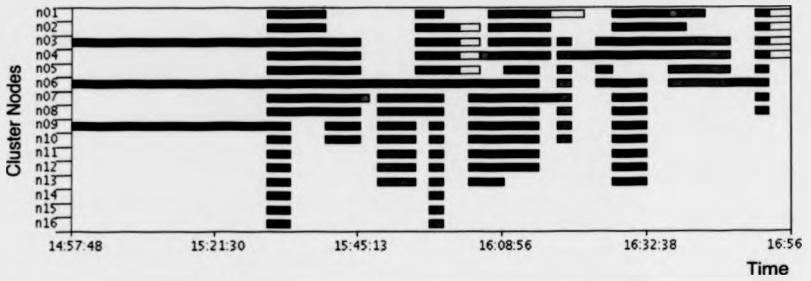


Figure 10.3: A scheduling time-line of 32 applications prior to refinement.

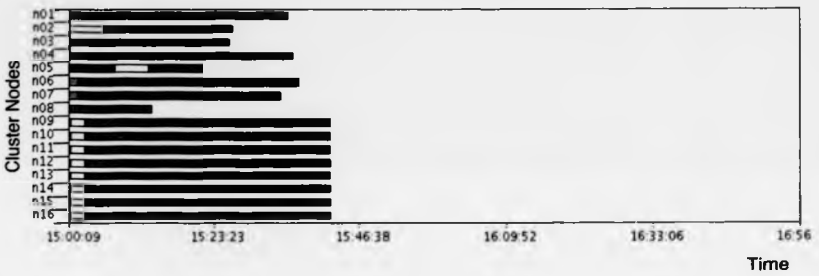


Figure 10.4: The scheduling time-line after 2000 heuristic iterations.

tures: 'C1' represents a 16-node cluster of Linux workstations, each containing a Pentium IV 2.4GHz processor with 512MBs RAM; 'C2' represents a 16-node cluster of Linux workstations that each contain a Pentium III 800MHz processor with 128MBs RAM; 'C3' represents a 16-node cluster of Solaris workstations that each contain an UltraSparcII 360MHz processor with 128MBs RAM. Each resource is inter-connected with a 100Mb Ethernet network.

Two benchmark applications were submitted every second to random agents within the agent hierarchy. Each submission contained the benchmark's jPACE performance characterisation as well as a random deadline and represents a user submitting an application to their local resource. Each application is initially queued successively on that agent's cluster using a 'first-come first-served' approach. Figure 10.6 illustrates the initial scheduling time-line of each agent within the hierarchy prior to any

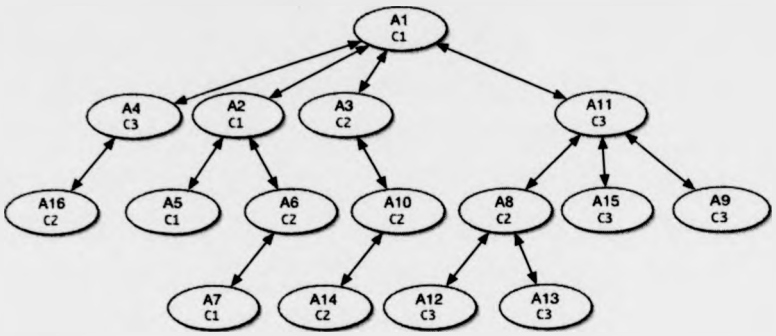


Figure 10.5: A TITAN agent hierarchy of a distributed, heterogeneous computing environment.

performance-based optimisation. Each line within this diagram represents the utilisation of that agent's cluster; the darker the colour at any one time, the greater the cluster utilisation. Due to each submission being made randomly, the percentage of tasks scheduled to be executed upon each agent (shown to the right of the diagram) is roughly uniform among each local resource.

Figure 10.7 illustrates the improvements brought about by inter-domain and cluster level performance optimisation. By using the predicted performance of each application on each of the resources, the agents are able to intelligently move tasks throughout the agent hierarchy. This results in a reduction in idle time among clusters and a 70% reduction in overall execution time from 4357 to 1342 seconds. This movement of tasks is demonstrated by agent '`A1`'s' execution of 20% of all submitted tasks; a result of it being one of the higher-performing clusters, and because of its location at the top of the agent hierarchy, which makes it more accessible to the other agents within the distributed computing environment.

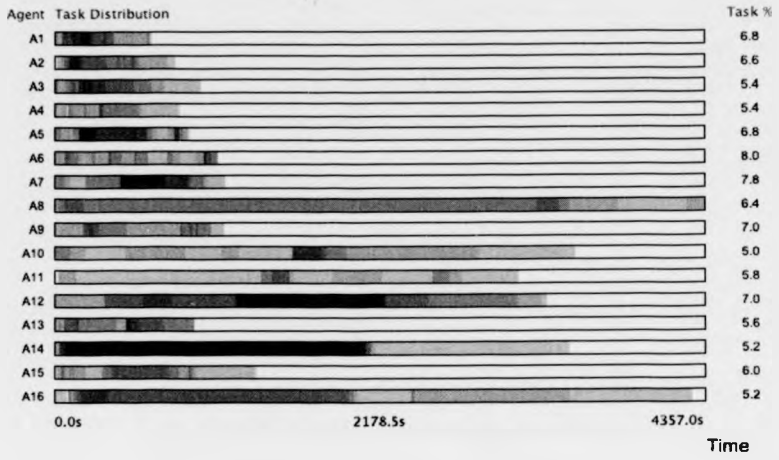


Figure 10.6: The original scheduling queue for each agent's resource prior to execution.

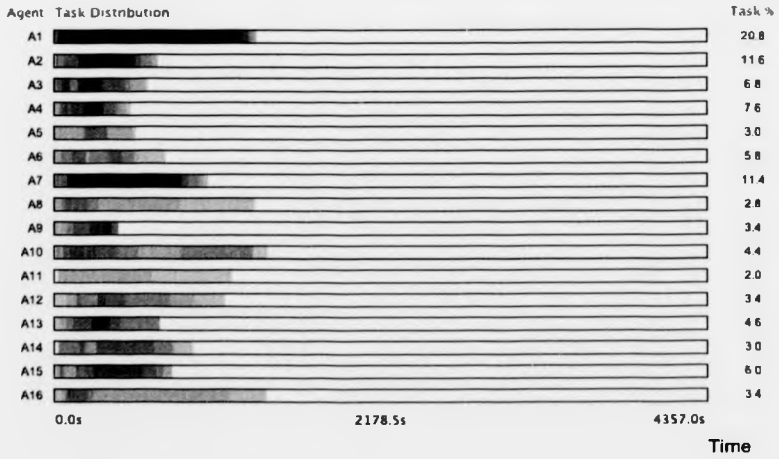


Figure 10.7: The scheduling queue as a result of both inter-domain and cluster-level performance optimisation.

10.2 Performance-based Web Service Routing

This section documents the service routing of requests within an example dynamic business-to-business (B2B) application called Gourmet2Go. The jPACE framework

is used to monitor and evaluate the performance of a number of Gourmet2Go's back-end web services. This performance data is then used by the Gourmet2Go broker in order to service route user-driven requests to the highest-performing service, without user-intervention. A number of scenarios are presented with a range of differently-performing services in order to illustrate this approach.

10.2.1 Gourmet2Go

Gourmet2Go is a web services demonstrator from the IBM Web Services Toolkit [IBM03b]. The demonstrator provides an example of how a web service acts as an intermediary broker, assisting users to select back-end web services by obtaining bids from services published in a UDDI registry [Project00]. In Gourmet2Go, the back-end web services sell groceries and the broker presents itself as a value-added 'meal planning' service. The underlying architecture is however generic, and is therefore used to demonstrate the brokering of any kind of service.

The architecture of the Gourmet2Go demonstrator can be found in Figure 10.8. A typical interaction is as follows:

1. The user interacts with the Gourmet2Go web application via a web browser with the intention of building a shopping list of groceries. This represents the user specifying the service that the back-end web service must perform.

2. The broker searches the registry for businesses with published web services that sell groceries. This represents the broker selecting a number of potential services using information provided in the registry.

3. The shopping list is passed by the broker to each of the back-end web services (located from the registry) via a '`getBid`' request. This represents the broker contacting the short-list of candidates directly.

4. The broker summarises the bids for the user based on price, and the user then

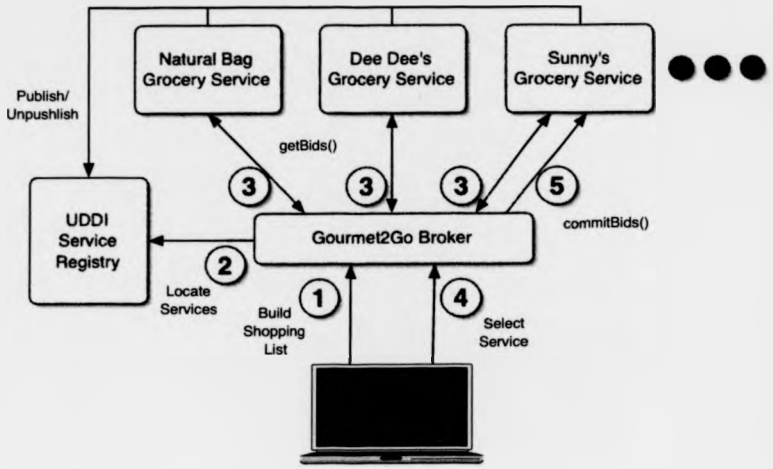


Figure 10.8: The design of the Gourmet2Go demonstrator from the IBM Web Services Toolkit. A typical interaction is shown in which a user browses the Gourmet2Go broker to obtain information on the available services and then selects the preferred service.

selects a supplier using the information presented to them. This stage represents the broker assisting the user in decision making.

5. The broker sends a '`commitBid`' request to the service that the user selects. This represents the broker continuing to act as an intermediary while the user interacts with the selected back-end service. This interaction has the potential to be significantly more complex than the confirmation message in the Gourmet2Go demonstrator. For example, specifying the details of what exactly is to be purchased and how it is to be paid for, through to delivery tracking and after-sales support.

10.2.2 Performance Evaluation

Gourmet2Go was chosen as the initial demonstrator in order to illustrate the use of an automated and performance-based service routing algorithm. Instead of the Gourmet2Go broker allowing the user to choose their preferred grocery service based on price, it

could be extended such that the service was automatically chosen based on performance, without user intervention. This implementation could form the basis of a service routing implementation that routes user-driven requests within an e-business environment based on the user's Service Level Agreement (SLA) and QoS requirements.

In order to modify the broker to automatically route requests based on performance, a measure of a service's performance must first be defined. The service's end-to-end response time is a good measure of performance and could be monitored during execution in order to evaluate a prediction. While the service routing algorithm is only used to automate which service's '`commitBid`' request to invoke, both the '`getBid`' and '`commitBid`' requests provide a good indication of the back-end service's performance. Therefore the average response times of both these requests are used to predict the next '`commitBid`' request's end-to-end response time. The evaluated end-to-end response time (ER) of a specific service is calculated by:

$$ER = (\frac{n_b}{n_t} * r_b) + (\frac{n_c}{n_t} * r_c) \tag{10.1}$$

where n_b, n_c and n_t are the number of '`getBid`', '`commitBid`' and total number of requests respectively, and r_b and r_c are the average response times of the '`getBid`' and '`commitBid`' requests respectively.

Due to fluctuations in the back-end service's end-to-end response times over time, a confidence metric must be associated with a service's response time. Requests can then be routed according to the confidence associated with the service's evaluated performance. It was decided to define the service's predicted confidence in relation to the number of times the broker sends a request to that service; ie. the number of previous '`getBid`' and '`commitBid`' service invocations. It was also decided to implement a so-called 'warm-up' feature for each service. The confidence of each request is set to zero until there has been a specific number of initial request invocations. Once the service's confidence rises above zero, a more representative average response time results. The evaluated confidences of a specific service's '`getBid`' (EC_b) and

'commitBid' requests (EC_c) were calculated by:

$$EC_b = \begin{cases} 0, & \text{if } n_b < n_m \\ n_b, & \text{otherwise} \end{cases} \tag{10.2}$$

$$EC_c = \begin{cases} 0, & \text{if } n_c < n_m \\ n_c, & \text{otherwise} \end{cases} \tag{10.3}$$

respectively, where n_m is the number of 'warm-up' requests. These two confidence values were then combined as illustrated below in order to calculate an evaluated confidence for the service. It was decided that, while this evaluated confidence should continue to rise over time, it should also slowly level off. To implement this feature, a logarithmic function was used such that the final calculation used to obtain a service's evaluated confidence (EC) was:

$$EC = log(\frac{1}{2}(C_b + 1) + \frac{1}{2}(C_c + 1)) \tag{10.4}$$

The 'Weighted Expected Performance' (WEP) of a particular service is calculated as:

$$WEP = \frac{EC}{ER} \tag{10.5}$$

such that a high evaluated confidence for a low evaluated end-to-end response time will result in a high WEP. WEP was used in order to choose which service was automatically chosen by the broker.

Six jPACE performance models were developed in order to evaluate the confidences and end-to-end response times of the 'getBid' and 'commitBid' requests on each of the three back-end services. These evaluated performances could then be used to obtain a service's WEP. Each model contained a simple transaction object, which defined a single 'variable', 'method' and 'confidence' declaration. The 'method' declaration is of type 'transaction', it references the method invoked by the broker in order to perform either the 'getBid' or 'commitBid' request, and its evaluated response time is obtained from the platform's historical data. The

```
2
3  <jPACE:transaction>
4
5    <jPACE:variable name="nm"/>
6
7    <jPACE:confidence variable="c">
8      <jPACE:if leftExpression="${nE}" condition="LESS_THAN"
9                rightExpression="${nm}">
10       <jPACE:setVariable variable="c" value="0"/>
11     </jPACE:if>
12     <jPACE:if leftExpression="${nE}" condition="GREATER_THAN_OR_EQUALS_TO"
13               rightExpression="${nm}">
14       <jPACE:setVariable variable="c" value="${nE}"/>
15     </jPACE:if>
16   </jPACE:confidence>
17
18   <jPACE:proc name="main">
19     <jPACE:evaluateMethod class="com.ibm.ews.g2gServices.
20                                  grocery.SammysGroceryService"
21                           method="getBid" descriptor="()V"/>
22   </jPACE:proc>
23
24   <jPACE:method class="com.ibm.ews.g2gServices.grocery.SammysGroceryService"
25                 method="getBid" descriptor="()V" type="transaction"/>
26
27 </jPACE:transaction>
28
```

Listing 10.1: The Sammy's grocery service's '`getBid`' request transaction object.

'`confidence`' declaration is used to associate a confidence with the evaluated transaction as defined previously. This confidence is dependent on the value of n_m, the number of iterations before the 'warm-up' period is finished, and is a parameter of the model. Listings 10.1 and 10.2 contain the two transaction objects of the '`Sammy's`' grocery service's '`getBid`' and '`commitBid`' request's performance models. The other four model's transaction objects, which evaluate the performance of the other two back-end service's requests, are the same as these except the '`method`' declarations reference their respective service methods.

The request's historical data was kept up-to-date within the broker's platform resource object using automated refinement. Each request's method was automatically instrumented as an ARM transaction as discussed in the previous chapter, and the appropriate '`methodTiming`' declaration's '`noExecutions`' and '`avResponse Time`' attributes were updated after each '`getBid`' and '`commitBid`' request. Every time the broker evaluated a model, the evaluated response time and confidence for that request were calculated from all previous historical data.

```
2
3  <jPACE:transaction>
4
5    <jPACE:variable name="nm"/>
6
7    <jPACE:confidence variable="c">
8      <jPACE:if leftExpression="${nE}" condition="LESS_THAN"
9                rightExpression="${nm}">
10       <jPACE:setVariable variable="c" value="0"/>
11     </jPACE:if>
12     <jPACE:if leftExpression="${nE}" condition="GREATER_THAN_OR_EQUALS_TO"
13               rightExpression="${nm}">
14       <jPACE:setVariable variable="c" value="${nE}"/>
15     </jPACE:if>
16   </jPACE:confidence>
17
18   <jPACE:proc name="main">
19     <jPACE:evaluateMethod class="com.ibm.ews.g2gServices.
20                                  grocery.SammysGroceryService"
21                           method="commitBid" descriptor="()V"/>
22   </jPACE:proc>
23
24   <jPACE:method class="com.ibm.ews.g2gServices.grocery.SammysGroceryService"
25                 method="commitBid" descriptor="()V" type="transaction"/>
26
27 </jPACE:transaction>
28
```

Listing 10.2: The Sammy's grocery service's '`commitBid`' request transaction object.

Once the models had been developed, the broker's source code was instrumented to automate the service routing of the '`commitBid`' requests based on their calculated performance. Where the broker originally presented each service's '`getBid`' results to the user via the web-based interface, each service's '`commitBid`' request was instead evaluated and each service's WEP was calculated as previously described. The '`commitBid`' request was then automatically sent to the service with the highest WEP. If this highest WEP was calculated for two or more services, one of them was chosen at random.

10.2.3 Service-routing Scenarios

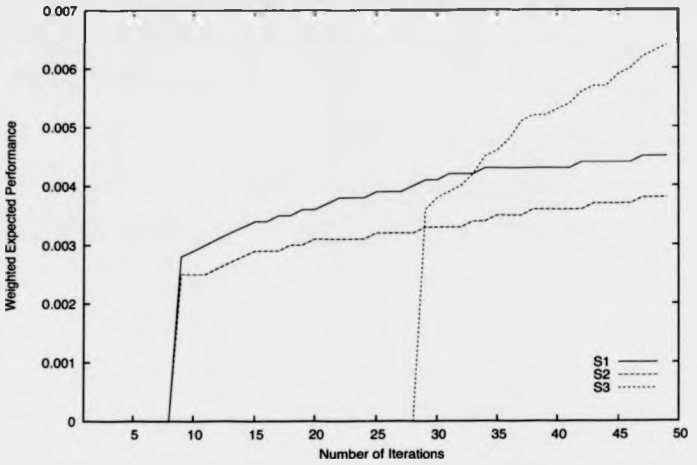
This performance-based service routing approach was tested for three different Gourmet-2Go scenarios. Each scenario starts with just two back-end services published within the UDDI registry, resulting in only these two services being routed to during a Gourmet-2Go iteration. After twenty iterations, the third service is published in order to illustrate

the effect of providing the application with a new back-end service. To facilitate the analysis of the results achieved from these scenarios, a number of enhancements were made to the Gourmet2Go simulation:

- A workload generator was written to interact with the broker's HTTP-based web service interface and invoke the same requests that a user-controlled browser would submit during a run-through of the Gourmet2Go demonstrator. This allows the demonstrator to automatically run through a large number of iterations in order to illustrate the service routing approach over time.

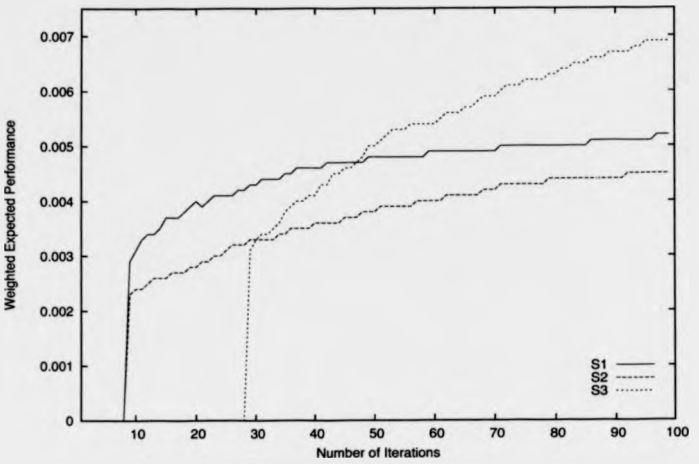
- The '`getBid`' and '`commitBid`' requests of the three back-end services were weighted with a random performance delay. Both the '`Natural Bag`' (S_1) and '`Sammy's`' (S_2) services were weighted equally such that their average response time was the same. These services represent the established service providers published at the start of each scenario. The '`Dee Dees`' (S_3) service represents the new service provider, and was weighted differently for each of the three scenarios. Each weighting incorporates a random element allowing a 40% deviation from the mean in end-to-end response time. This is in order to simulate a number of likely performance-hindering factors including communication delay, data-dependencies and the inconsistent load on both the broker and back-end servers.

The results obtained from each scenario are presented in Graphs 10.1, 10.2 and 10.3. Each graph illustrates the range of WEP for each back-end service over the course of a certain number of Gourmet2Go iterations. The service with the largest WEP at any one iteration will be sent the '`commitBid`' request by the service routing algorithm. S_1 and S_2 are both published in the service registry at the start of each simulation. S_3 is weighted at three different levels ranging from a much higher performance than the other two services to the same performance, and is published after twenty iterations.

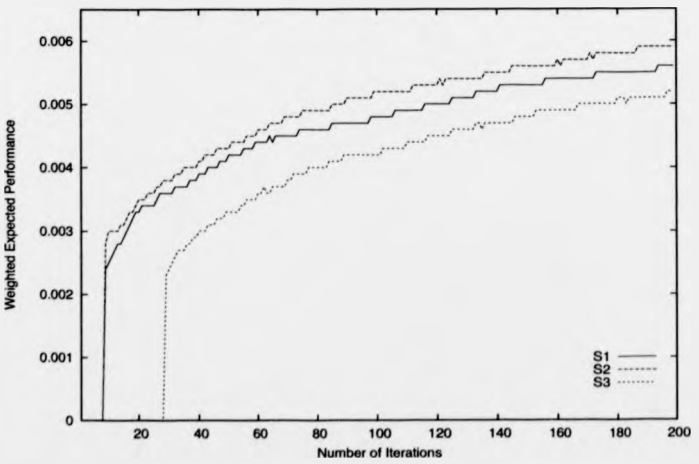


Graph 10.1: The simulation results when S_3 is published as a 'high performance' service in relation to S_1 and S_2. The results show that S_1 is initially selected over S_2, due to the fact that it performed better during the 'warm-up' period. It then continues to be used because of its increased confidence over S_2; the broker picks a service that seems to perform well and continues to use it whilst it performance consistently. After twenty iterations, the service S_3 is published. S_3 provides '`getBid`' response times throughout its 'warm-up' period even though it is not selected by the broker due to its evaluated confidence of zero. Once available for selection, its superior performance provides a sharp increase in WEP, ensuring its dominance for the remainder of the scenario.

The service routing algorithm described above is a first step into the balancing of requests among an e-business framework with the use of performance data. It follows from these results that a new web service would have to perform significantly better than the currently published services for the broker to select this new service over those with a large evaluated confidence. This implementation of the routing algorithm would simply send all requests to the service which is deemed the highest-performing, rather than intelligently balancing the workload among all services. A more sophisticated service routing framework is in development and is not part of this research.



Graph 10.2: The simulation results when S_3 is published as a service of higher performance than S_1 and S_2. The results are similar to Figure 10.1, except here it takes longer for S_3 to become the more dominant service. Again, once selected, the confidence in service S_3 increases and it continues to be chosen as the preferred service.



Graph 10.3: The simulation results when S_3 is published with the same performance as S_1 and S_2. Unless the average response time for S_3 improves over the other services, its confidence will never overtake that of S_2, for which '`commmitBid`' and '`getBid`' requests are being recorded. S_2 is the preferred service throughout the simulation due to its better performance during 'warm-up'.

10.3 Summary

This chapter provided an overview of two performance-based middleware services that are currently being developed at the University of Warwick. Each service uses predicted performance information obtained from the jPACE evaluation engine in order to improve the efficiency of task scheduling and request routing. TITAN, a Grid resource scheduler, efficiently maps scientific applications among a heterogeneous, distributed environment by evaluating the application's predicted performance on the available resources prior to its execution. This performance data allows TITAN to move submitted applications within the environment so that deadline times are maintained, the overall load on the system is balanced and the makespan is reduced. Also documented was a B2B service routing algorithm that is used to automate the flow of transaction-based requests among published web services. The jPACE prediction and monitoring framework is used to provide an evaluated end-to-end response time and confidence for these web services, which can then be used to automatically route requests to the highest-performing service.

The development of these performance-based middleware services is ongoing. TITAN is currently being extended so that a workflow of applications, whose successful completion is dependent on the results of each other's execution, can be efficiently scheduled within a heterogeneous environment [Cao03]. The B2B service routing framework is currently being extended as a Dynamic Workload Management infrastructure [Bacigalupo03b] that aims to meet user-driven Service Level Agreements (SLAs). A jPACE platform implementation of a generic e-business system model is also in development.

Chapter 11

Conclusion

Modern, high-performance computing environments can be very complex. Such environments currently include massively-parallel cluster-based architectures and heterogeneous, geographically-spaced and resource-sharing environments such as those proposed within the Grid computing paradigm. While these platforms can theoretically represent a vast amount of computing power, making best use of this power is particularly difficult, especially if user-driven quality of service levels are to be maintained. With the presence of accurate performance data, applications can be intelligently placed within the environment such that their deadlines are guaranteed. Furthermore, predicting an application's performance can reduce the idle time of resources, resulting in a higher average throughput and CPU utilisation.

This thesis documents the implementation of a dynamic prediction and monitoring framework, known as jPACE, which can be used to provide middleware services with the performance data required to improve the environment's overall efficiency. The jPACE framework is an extension of the original PACE predictive framework. Original PACE performance models are specialised and inflexible, as they often require manual refinement before accurate predictive evaluations can be achieved. Furthermore, they do not capture the performance of modern execution environments, such as virtual machines and web-based e-business frameworks, for example. A number

of extensions to PACE have been documented that provide a more dynamic, flexible and automated approach to performance characterisation and evaluation for modern computing applications and execution environments.

One such extension has been the development of a more flexible and portable characterisation language. This language, known as the jPACE Performance Characterisation Language (jPCL) is transaction-based, allowing the performance of a broader class of application to be described. Each transaction can capture the performance of any amount of work, both computation and communication, thus facilitating the characterisation of applications at varying levels of abstraction and complexity. This enables the model developer to make a trade-off between characterisation development time and the eventual predictive accuracy. The Automated Characterisation Tool (ACT), a sophisticated tool that is used to automate the development of transactions, has also been developed.

PACE has been further extended to facilitate the characterisation of modern heterogeneous execution environments. Within jPACE a platform interface is employed, with a specific type of execution environment characterised by implementing this platform interface. During evaluation this interface is accessed in order to predict the execution environment's performance. This interface implementation is responsible for the modelling of runtime performance variations, and is supported by a number of jPCL performance objects that contain benchmark timings for each element of computation and communication. The development of a Hotspot Java Virtual Machine platform interface implementation that models a collection of runtime optimisations that occur during execution, has been documented in this thesis.

A parametric evaluation engine has been developed in order to evaluate the performance of an application that has been characterised in jPCL on a modelled execution environment. This evaluation engine was implemented in Java for portability, and facilitates the data and scalability analysis of applications on a given distributed execution environment. The jPACE evaluation engine also extends the original PACE evalua-

tion engine by assigning an associated confidence to predictions. This confidence can be used by middleware services in order to more efficiently manage an application's execution.

A number of JavaGrande benchmarks have been characterised and evaluated in order to compare the benchmark's actual measured performance with the predicted performance that is obtained by the jPACE framework. A good predictive accuracy is achieved that ranges from 1% to 30% for the applications documented in this thesis. This is comparable with the original PACE performance models that have a similar inaccuracy.

An application monitoring framework for the automated refinement of performance characterisations has also been described in this thesis. An extension to the Application Response Measurement (ARM) standard has been employed in order to measure unpredictable elements of the application during execution. This historical data is used by the evaluation engine to refine both an application's jPCL characterisation and also the platform's historical data in order to achieve more accurate predictive evaluations in the future. The performance characterisation of further data-dependent applications has also been documented. This refinement results in an increase in both predictive accuracy and also the associated confidence after each application execution.

The use of the jPACE framework within two performance-based middleware services has been documented, highlighting the use of predictive performance services within complex computing environments. Each service aims to increase the environment's efficiency and overall performance with the use of accurate predictive performance data. The jPACE prediction and monitoring framework documented in this thesis has been used to provide each of these services with this performance data. Results show that employing this performance data can improve the schedule of common scientific tasks on a distributed, heterogeneous computing environment by up to 70%.

The research documented in this thesis has contributed to the fields of Grid computing, performance modelling and the Application Response Measurement (ARM)

standard. While the analytical modelling techniques developed in PACE and within this thesis are similar to that of POEMS [Deelman98], POEMS does not currently support the characterisation and prediction of Java applications. The development of jPCL contributes to the Grid computing community a powerful, flexible and portable XML-based language for describing the performance of applications. Furthermore, the automated bytecode instrumentation of ARM consumer interface API calls as documented in Chapter 9 and [Turner01] has contributed a tool for the quick and efficient 'ARMing' of Java applications.

11.1 Future Work

There are a number of areas of future work. Three such areas that were illustrated during the course of this thesis were:

1. *Extending the Functionality of the Automated Characterisation Tool.* The documentation in Chapter 5 of the Automated Characterisation Tool (ACT) illustrated a number of limitations in the ACT's functionality. The ACT cannot currently calculate: the probabilities of parameter-dependent conditional statements within an application; a number of more complicated iterative expressions where it is not easy to calculate the changes of the loop's control variable(s) after each iteration; the value of a parameter-dependent variables from methods not defined within the transaction being characterised. The ACT's functionality could be extended by implementing more of the same techniques currently used by the ACT such that when these more complicated situations occur during characterisation (an iterative statement consisting of more than one control variable for example) they can be processed and the parameter-dependent variables calculated accordingly. These extensions to the ACT's functionality would help to further reduce the time necessary in order to achieve accurate predictive evaluations.

2. *A More In-Depth Characterisation of the Java Virtual Machine.* As discussed in Chapter 6, elements of the performance-critical runtime behaviour of the Java Virtual Machine are characterised within the platform implementation's virtual machine performance object. However, only the JVM's adaptive compilation of bytecode is currently modelled within this performance object and other elements such as garbage collection, heap allocation and monitor contention are currently not characterised. If further insight into how these elements affect the performance of a Java application during execution was achieved, they could be characterised, included within the platform's virtual machine object and taken into account during the evaluation of Java applications in much the same way as the JVM's adaptive compilation is currently evaluated. Implementing these extensions would result in the more accurate performance prediction of Java applications.

3. *The Automated Processing of an Application's Data during Evaluation.* It was documented in Chapter 3 how the performance of a data-dependent application could be accurately predicted if the application's data was first processed prior to evaluation. Processing the data in this way allows the performance-critical nature of the data to be recognised and data-dependent elements of the model to be accurately evaluated. However this processing requires an external application to be written that specifically looks at the correct elements of the data that affect the application's performance.

   It is currently deemed possible that the evaluation engine could be extended such that the automated processing of data in this way could be performed automatically during evaluation, without the requirement of constructing this external application. However, the implementation of such a technique would be extremely complex. In a similar way that the ACT calculates parameter-dependent variables by scanning through the application's previous execution, the ACT could

also calculate specifically which data is being accessed and assign this data to data-dependent characterisations within the model. During an evaluation, this data could then be processed and data-dependent variables be accurately calculated accordingly. Implementing this extension would dramatically reduce the time currently required to accurately predict the performance of data-dependent applications prior to execution and automated refinement.

Furthermore, there are a number of more long-term areas of work which illustrate my insight into how the future development of this research should proceed. These fall into three distinct categories:

1. *The Characterisation of a Broader Class of Application.* jPCL has been developed as a transaction-based language in order to provide a flexible and dynamic framework for performance characterisation. jPCL could be extended so that other types of application could be characterised. These potentially include: compiled applications that execute natively, such as those characterised by PACE; other virtual machine-based execution environments, such as .NET; user-driven service-based e-business applications; the five proposed classes of Grid applications as discussed in Chapter 1. Each new class of application would require an extension to the current XML-based syntax in order to capture the application's control flow of performance-critical elements during execution. For example, the extension of jPCL in order to characterise Data-Intensive Computing applications (which involves the synthesis of new information from many or large data sources [Foster98]) would require the development of an XML syntax in order to describe the performance of distributed database access to and from remote data repositories.

2. *The Characterisation of a Broader Class of Execution Environment.* The addition of another type or class of application to the jPCL syntax would also require the development of an associated platform interface implementation. This imple-

mentation would model the specific runtime performance variance that can result during that class of application's execution. Two examples of platform-specific elements that are currently not characterised within jPCL are the continuing variance of web server and database loads within an e-business framework and the inter-domain wide-area communication of large amounts of data throughout a Grid environment. However, predicting this inter-domain communication is a complicated task as the fluctuations of network performance that generally occur are very difficult to evaluate. Such an execution environment's implementation may therefore be forced to rely on existing network modelling equations such as the Network Weather Service (NWS) [Wolski99] during an evaluation if accurate predictive evaluations are to be achieved.

3. *Grid Standards Integration.* The Grid research community is currently engaged in the development of a number of common standards around which a Grid environment will flourish. Globus is a forerunner in this research, and the middleware services being developed at Warwick are being integrated into the Globus framework. It would be prudent to link the jPACE framework into the Globus information services in order to create a 'Grid-enabled' performance service. These information services could contain historical performance data as reported by ARM, as well as complete jPCL performance characterisations.

Appendix A

JavaGrande Benchmark Source Code

The following twenty-two pages contain the source code for the five JavaGrande benchmarks whose characterisation was documented in this thesis. The Sparse Matrix Multiply benchmark is implemented by the '`JGFSparseMatmultBench`' and '`Sparse Matmult`' classes. The Fourier Coefficient Analysis benchmark is implemented by the '`JGFSeriesBench`' and '`SeriesTest`' classes. The IDEA Encryption benchmark is implemented by the '`JGFCryptBench`' and '`IDEATest`' classes. The Gaussian Random Number Generation benchmark is implemented by the '`DHPC_EP Bench`' and '`KernelEP`' classes. The Fast Fourier Transform benchmark is implemented by the single '`DHPC_FFTBench`' class.

```
/**************************************************************************
 *                                                                        *
 *             Java Grande Forum Benchmark Suite - MPJ Version 1.0        *
 *                                                                        *
 *                                produced by                             *
 *                                                                        *
 *                    Java Grande Benchmarking Project                    *
 *                                                                        *
 *                                   at                                   *
 *                                                                        *
 *                  Edinburgh Parallel Computing Centre                   *
 *                                                                        *
 *                  email: epcc-javagrande@epcc.ed.ac.uk                  *
 *                                                                        *
 *                                                                        *
 *      This version copyright (c) The University of Edinburgh, 2001.     *
 *                         All rights reserved.                           *
 *                                                                        *
 **************************************************************************/


package uk.ac.warwick.dcs.hpsg.applications.jgf.sparsematmult;

import mpi.*;

public class JGFSparseMatmultBench extends SparseMatmult {

    public static int nprocess;
    public static int rank;
    private int size;
    private static final long RANDOM_SEED = 10101010;

    private static final int datasizes_M[] = {/*10000,*/ 200000, 500000, 800000}
;
    private static final int datasizes_N[] = {/*10000,*/ 200000, 500000, 800000}
;
    private static final int datasizes_nz[] = {/*50000,*/ 1000000, 2500000, 6400
000};
    private static final int SPARSE_NUM_ITER = 200;

    Random R = new Random(RANDOM_SEED);

    double [] x;
    double [] y;
    double [] p_y;
    double [] val = null;
    int [] col = null;
    int [] row = null;
    double [] buf_val = null;
    int [] buf_col = null;
    int [] buf_row = null;
```



```
    int p_datasizes_nz,ref_p_datasizes_nz,rem_p_datasizes_nz;

    public JGFSparseMatmultBench(int nprocess, int rank) {
        this.nprocess=nprocess;
        this.rank=rank;
    }

    public void JGFsetsize(int size){
        this.size = size;
    }

    public void JGFinitialise() throws MPIException{

        /* Determine the size of the arrays row,val and col on each
           process. Note that the array size on process (nprocess-1) may
           be smaller than the other array sizes.
        */

        p_datasizes_nz = (datasizes_nz[size] + nprocess - 1) / nprocess;
        ref_p_datasizes_nz = p_datasizes_nz;
        rem_p_datasizes_nz = p_datasizes_nz - ((p_datasizes_nz*nprocess) - datas
izes_nz[size]);

        if(rank==(nprocess-1)){
            if((p_datasizes_nz*(rank+1)) > datasizes_nz[size]) {
                p_datasizes_nz = rem_p_datasizes_nz;
            }
        }

        /* Initialise the arrays val,col,row. Create full sizes arrays on proces
s 0 */

        x = RandomVector(datasizes_N[size], R);
        y = new double[datasizes_M[size]];
        p_y = new double[datasizes_M[size]];

        val = new double[p_datasizes_nz];
        col = new int[p_datasizes_nz];
        row = new int[p_datasizes_nz];

        if(rank==0) {
            buf_val = new double[datasizes_nz[size]];
            buf_col = new int[datasizes_nz[size]];
            buf_row = new int[datasizes_nz[size]];
        }

        /* initialise arrays val,col,row on process 0 and send the data to
           the other processes
        */

        if(rank==0) {
```

```
            for (int i=0; i<p_datasizes_nz; i++) {

                // generate random row index (0, M-1)
                row[i] = Math.abs(R.nextInt()) % datasizes_M[size];
                buf_row[i] = row[i];
                // generate random column index (0, N-1)
                col[i] = Math.abs(R.nextInt()) % datasizes_N[size];
                buf_col[i] = col[i];
                val[i] = R.nextDouble();
                buf_val[i] = val[i];
            }

            for(int k=1;k<nprocess;k++) {
                if(k==nprocess-1) {
                    p_datasizes_nz = rem_p_datasizes_nz;
                }
                for (int i=0; i<p_datasizes_nz; i++) {
                    buf_row[i+(k*ref_p_datasizes_nz)] = Math.abs(R.nextInt()) %
datasizes_M[size];
                    buf_col[i+(k*ref_p_datasizes_nz)] = Math.abs(R.nextInt()) %
datasizes_N[size];
                    buf_val[i+(k*ref_p_datasizes_nz)] = R.nextDouble();
                }
                MPI.COMM_WORLD.Ssend(buf_row,(k*ref_p_datasizes_nz),p_datasizes_
nz,MPI.INT,k,1);
                MPI.COMM_WORLD.Ssend(buf_col,(k*ref_p_datasizes_nz),p_datasizes_
nz,MPI.INT,k,2);
                MPI.COMM_WORLD.Ssend(buf_val,(k*ref_p_datasizes_nz),p_datasizes_
nz,MPI.DOUBLE,k,3);
            }

            p_datasizes_nz = ref_p_datasizes_nz;

        } else {
            MPI.COMM_WORLD.Recv(row,0,p_datasizes_nz,MPI.INT,0,1);
            MPI.COMM_WORLD.Recv(col,0,p_datasizes_nz,MPI.INT,0,2);
            MPI.COMM_WORLD.Recv(val,0,p_datasizes_nz,MPI.DOUBLE,0,3);
        }

    }

    public void JGFkernel() throws MPIException{

        SparseMatmult.test(y, val, row, col, x, SPARSE_NUM_ITER, buf_row, p_y);

    }

    public void JGFvalidate(){

        if(rank==0) {
            double refval[] = {75.02484945753453,150.0130719633895,749.524587075
3752};
```



```
            double dev = Math.abs(ytotal - refval[size]);
            if (dev > 1.0e-12 ){
                System.out.println("Validation failed");
                System.out.println("ytotal=" + ytotal + " " + dev + " " + size);
            }
        }

    }

    public void JGFtidyup(){
        System.gc();
    }

    public void JGFrun(int size) throws MPIException{

        JGFsetsize(size);
        JGFinitialise();
        JGFkernel();
        JGFvalidate();
        JGFtidyup();

    }

    private static double[] RandomVector(int N, java.util.Random R)
    {
        double A[] = new double[N];

        for (int i=0; i<N; i++)
            A[i] = R.nextDouble() * 1e-6;

        return A;
    }

}
```

```
/**************************************************************************
 *                                                                        *
 *             Java Grande Forum Benchmark Suite - MPJ Version 1.0        *
 *                                                                        *
 *                                produced by                             *
 *                                                                        *
 *                    Java Grande Benchmarking Project                    *
 *                                                                        *
 *                                    at                                  *
 *                                                                        *
 *                  Edinburgh Parallel Computing Centre                   *
 *                                                                        *
 *                  email: epcc-javagrande@epcc.ed.ac.uk                  *
 *                                                                        *
 *      adapted from SciMark 2.0, author Roldan Pozo (pozo@cam.nist.gov)  *
 *                                                                        *
 *      This version copyright (c) The University of Edinburgh, 2001.     *
 *                         All rights reserved.                           *
 *                                                                        *
 **************************************************************************/

package uk.ac.warwick.dcs.hpsg.applications.jgf.sparsematmult;

import mpi.*;

public class SparseMatmult
{

    public static double ytotal = 0.0;

    /* 10 iterations used to make kernel have roughly
       same granulairty as other Scimark kernels. */

    public static void test( double y[], double val[], int row[],
                             int col[], double x[], int NUM_ITERATIONS, int buf_
row[], double p_y[]) throws MPIException
    {
        int nz = val.length;

        MPI.COMM_WORLD.Barrier();

        for (int reps=0; reps<NUM_ITERATIONS; reps++)
            {
                for (int i=0; i<nz; i++)
                    {
                        p_y[ row[i] ] += x[ col[i] ] * val[i];
                    }
            }

        // create an updated copy on each process
        MPI.COMM_WORLD.Allreduce(p_y,0,y,0,y.length,MPI.DOUBLE,MPI.SUM);
```



```
        MPI.COMM_WORLD.Barrier();

        if(JGFSparseMatmultBench.rank==0){
            for (int i=0; i<buf_row.length; i++) {
                ytotal += y[ buf_row[i] ];
            }
        }

    }
}
```

```
/**************************************************************************
 *                                                                        *
 *             Java Grande Forum Benchmark Suite - MPJ Version 1.0        *
 *                                                                        *
 *                                produced by                             *
 *                                                                        *
 *                    Java Grande Benchmarking Project                    *
 *                                                                        *
 *                                    at                                  *
 *                                                                        *
 *                    Edinburgh Parallel Computing Centre                 *
 *                                                                        *
 *                    email: epcc-javagrande@epcc.ed.ac.uk                *
 *                                                                        *
 *                                                                        *
 *      This version copyright (c) The University of Edinburgh, 2001.    *
 *                         All rights reserved.                           *
 *                                                                        *
 **************************************************************************/


package uk.ac.warwick.dcs.hpsg.applications.jgf.series;

import mpi.*;

public class JGFSeriesBench extends SeriesTest {

    public static int nprocess;
    public static int rank;
    private int size;
    private int datasizes[]={10000,50000,100000};

    public JGFSeriesBench(int nprocess, int rank) {
        this.nprocess=nprocess;
        this.rank=rank;
    }

    public void JGFsetsize(int size){
        this.size = size;
    }

    public void JGFinitialise(){
        array_rows = datasizes[size];

        /* determine the array dimension size on each process
           p_array_rows will be smaller on process (nprocess-1).
           ref_p_array_rows is the size on all processes except process (nproces
s-1),
           rem_p_array_rows is the size on process (nprocess-1).
        */

        p_array_rows = (array_rows + nprocess -1) / nprocess;
```



```
        ref_p_array_rows = p_array_rows;
        rem_p_array_rows = p_array_rows - ((p_array_rows*nprocess) - array_rows)
;
        if(rank==(nprocess-1)){
            if((p_array_rows*(rank+1)) > array_rows) {
                p_array_rows = rem_p_array_rows;
            }
        }

        buildTestData();
    }

    public void JGFkernel() throws MPIException{

        Do();

    }

    public void JGFvalidate(){
        double ref[][] = {{2.8729524964837996, 0.0},
                          {1.1161046676147888, -1.8819691893398025},
                          {0.34429060398168704, -1.1645642623320958},
                          {0.15238898702519288, -0.8143461113044298}};
        /*
        // for 200 points
        double ref[][] = {{2.8377707562588803, 0.0},
        {1.0457844730995536, -1.8791032618587762},
        {0.27410022422635033, -1.158835123403027},
        {0.08241482176581083, -0.8057591902785817}};
        */

        if(rank==0) {
            for (int i = 0; i < 4; i++){
                for (int j = 0; j < 2; j++){
                    double error = Math.abs(TestArray[j][i] - ref[i][j]);
                    if (error > 1.0e-12 ){
                        System.out.println("Validation failed for coefficient " + j + "," + i
);
                        System.out.println("Computed value = " + TestArray[j][i]);
                        System.out.println("Reference value = " + ref[i][j]);
                    }
                }
            }
        }
    }

    public void JGFtidyup(){
        freeTestData();
    }

    public void JGFrun(int size) throws MPIException{
```

```
        JGFsetsize(size);
        JGFinitialise();
        JGFkernel();
        JGFvalidate();
        JGFtidyup();

    }
}
```

```
/**************************************************************************
 *                                                                        *
 *             Java Grande Forum Benchmark Suite - MPJ Version 1.0        *
 *                                                                        *
 *                                produced by                             *
 *                                                                        *
 *                    Java Grande Benchmarking Project                    *
 *                                                                        *
 *                                   at                                   *
 *                                                                        *
 *                  Edinburgh Parallel Computing Centre                   *
 *                                                                        *
 *                  email: epcc-javagrande@epcc.ed.ac.uk                  *
 *                                                                        *
 *                    Original version of this code by                    *
 *                   Gabriel Zachmann (zach@igd.fhg.de)                   *
 *                                                                        *
 *      This version copyright (c) The University of Edinburgh, 2001.     *
 *                          All rights reserved.                          *
 *                                                                        *
 **************************************************************************/

/**
 * Class SeriesTest
 *
 * Performs the transcendental/trigonometric portion of the
 * benchmark. This test calculates the first n fourier
 * coefficients of the function (x+1)^x defined on the interval
 * 0,2 (where n is an arbitrary number that is set to make the
 * test last long enough to be accurately measured by the system
 * clock). Results are reported in number of coefficients calculated
 * per sec.
 *
 * The first four pairs of coefficients calculated shoud be:
 * (2.83777, 0), (1.04578, -1.8791), (0.2741, -1.15884), and
 * (0.0824148, -0.805759).
 */

package uk.ac.warwick.dcs.hpsg.applications.jgf.series;

import mpi.*;

class SeriesTest
{

    // Declare class data.

    int array_rows;
    int p_array_rows;
    int ref_p_array_rows;
    int rem_p_array_rows;
    double [] [] p_TestArray = null;  // Array of arrays on each process.
```



```
    double [] [] TestArray = null;  // Array of arrays.

    /*
     * buildTestData
     *
     */

    // Instantiate array(s) to hold fourier coefficients.

    void buildTestData()
    {
        // Allocate appropriate length for the double array of doubles.

        if(JGFSeriesBench.rank==0) {
            TestArray = new double [2][array_rows];
        }
        p_TestArray = new double [2][p_array_rows];
    }

    /*
     * Do
     *
     * This consists of calculating the
     * first n pairs of fourier coefficients of the function (x+1)^x on
     * the interval 0,2. n is given by array_rows, the array size.
     * NOTE: The # of integration steps is fixed at 1000.
     */

    void Do() throws MPIException
    {
        double omega;       // Fundamental frequency.
        int ilow;

        if(JGFSeriesBench.rank==0) {

            // Calculate the fourier series. Begin by calculating A[0].

            TestArray[0][0]=TrapezoidIntegrate((double)0.0, // Lower bound.
                                               (double)2.0,            // Upper
bound.
                                               1000,                    // # of
steps.
                                               (double)0.0,            // No ome
ga*n needed.
                                               0) / (double)2.0;       // 0 = te
rm A[0].
        }

        // Calculate the fundamental frequency.
        // ( 2 * pi ) / period...and since the period
        // is 2, omega is simply pi.
```

```
        omega = (double) 3.1415926535897932;

        if(JGFSeriesBench.rank==0) {
            ilow = 1;
        } else {
            ilow = 0;
        }

        for (int i = ilow; i < p_array_rows; i++)
            {
                // Calculate A[i] terms. Note, once again, that we
                // can ignore the 2/period term outside the integral
                // since the period is 2 and the term cancels itself
                // out.

                p_TestArray[0][i] = TrapezoidIntegrate((double)0.0,
                                                       (double)2.0,
                                                       1000,
                                                       omega * ((double)i + (ref
_p_array_rows*JGFSeriesBench.rank)),
                                                       1);
 // 1 = cosine term.

                // Calculate the B[i] terms.

                p_TestArray[1][i] = TrapezoidIntegrate((double)0.0,
                                                       (double)2.0,
                                                       1000,
                                                       omega * ((double)i + (ref
_p_array_rows*JGFSeriesBench.rank)),
                                                       2);
 // 2 = sine term.
            }

        MPI.COMM_WORLD.Barrier();

        // Send all the data to process 0

        if(JGFSeriesBench.rank==0) {

            for(int k=1;k<p_array_rows;k++){
                TestArray[0][k] = p_TestArray[0][k];
                TestArray[1][k] = p_TestArray[1][k];
            }

            for(int k=1;k<JGFSeriesBench.nprocess;k++) {

                MPI.COMM_WORLD.Recv(p_TestArray[0],0,p_TestArray[0].length,MPI.D
OUBLE,k,k);
                MPI.COMM_WORLD.Recv(p_TestArray[1],0,p_TestArray[1].length,MPI.D
OUBLE,k,
                                    k+JGFSeriesBench.nprocess);
```



```
                if(k==(JGFSeriesBench.nprocess-1)) {
                    p_array_rows = rem_p_array_rows;
                }

                for(int j=0;j<p_array_rows;j++){
                    TestArray[0][j+(ref_p_array_rows*k)] = p_TestArray[0][j];
                    TestArray[1][j+(ref_p_array_rows*k)] = p_TestArray[1][j];
                }

            }

            p_array_rows = ref_p_array_rows;

        } else {

            MPI.COMM_WORLD.Ssend(p_TestArray[0],0,p_TestArray[0].length,MPI.DOUB
LE,0,JGFSeriesBench.rank);
            MPI.COMM_WORLD.Ssend(p_TestArray[1],0,p_TestArray[1].length,MPI.DOUB
LE,0,
                                 JGFSeriesBench.rank+JGFSeriesBench.nprocess);
        }

        MPI.COMM_WORLD.Barrier();

    }

    /*
     * TrapezoidIntegrate
     *
     * Perform a simple trapezoid integration on the function (x+1)**x.
     * x0,x1 set the lower and upper bounds of the integration.
     * nsteps indicates # of trapezoidal sections.
     * omegan is the fundamental frequency times the series member #.
     * select = 0 for the A[0] term, 1 for cosine terms, and 2 for
     * sine terms. Returns the value.
     */

    private double TrapezoidIntegrate (double x0,     // Lower bound.
                                       double x1,     // Upper bound.
                                       int nsteps,    // # of steps.
                                       double omegan, // omega * n.
                                       int select)    // Term type.
    {
        double x;              // Independent variable.
        double dx;             // Step size.
        double rvalue;         // Return value.

        // Initialize independent variable.

        x = x0;
```

```
        // Calculate stepsize.

        dx = (x1 - x0) / (double)nsteps;

        // Initialize the return value.

        rvalue = thefunction(x0, omegan, select) / (double)2.0;

        // Compute the other terms of the integral.

        if (nsteps != 1)
            {
                --nsteps;               // Already done 1 step.
                while (--nsteps > 0)
                    {
                        x += dx;
                        rvalue += thefunction(x, omegan, select);
                    }
            }

        // Finish computation.

        rvalue=(rvalue + thefunction(x1,omegan,select) / (double)2.0) * dx;
        return(rvalue);
    }

    /*
     * thefunction
     *
     * This routine selects the function to be used in the Trapezoid
     * integration. x is the independent variable, omegan is omega * n,
     * and select chooses which of the sine/cosine functions
     * are used. Note the special case for select=0.
     */

    private double thefunction(double x,      // Independent variable.
                               double omegan,             // Omega * term.
                               int select)                // Choose type.
    {

        // Use select to pick which function we call.

        double returnValue;

        switch(select)
            {
            case 0: { returnValue = Math.pow(x+(double)1.0,x); break; }

            case 1: { returnValue = Math.pow(x+(double)1.0,x) * Math.cos(omegan*
x); break; }

            case 2: { returnValue = Math.pow(x+(double)1.0,x) * Math.sin(omegan*
```



```
x); break; }

                // We should never reach this point, but the following
                // keeps compilers from issuing a warning message.

            default: returnValue = 0;
            }

        return returnValue;
    }

    /*
     * freeTestData
     *
     * Nulls array that is created with every run and forces garbage
     * collection to free up memory.
     */

    void freeTestData()
    {
        TestArray = null;     // Destroy the array.
        System.gc();          // Force garbage collection.
    }

}
```

```
/**************************************************************************
 *                                                                        *
 *             Java Grande Forum Benchmark Suite - MPJ Version 1.0        *
 *                                                                        *
 *                                produced by                             *
 *                                                                        *
 *                    Java Grande Benchmarking Project                    *
 *                                                                        *
 *                                    at                                  *
 *                                                                        *
 *                  Edinburgh Parallel Computing Centre                   *
 *                                                                        *
 *                  email: epcc-javagrande@epcc.ed.ac.uk                  *
 *                                                                        *
 *                                                                        *
 *      This version copyright (c) The University of Edinburgh, 2001.     *
 *                         All rights reserved.                           *
 *                                                                        *
 **************************************************************************/


package uk.ac.warwick.dcs.hpsg.applications.jgf.crypt;

import mpi.*;

public class JGFCryptBench extends IDEATest {

    public static int nprocess;
    public static int rank;
    private int size;
    private int datasizes[] = { 5000000, 40000000, 80000000 };
    //  private int datasizes[]={5000000,20000000,50000000,80000000};

    public JGFCryptBench(int nprocess, int rank) {
        this.nprocess=nprocess;
        this.rank=rank;
    }

    public void JGFsetsize(int size){
        this.size = size;
    }

    public void JGFinitialise(){
        array_rows = datasizes[size];

        /* determine the array dimension size on each process
           p_array_rows will be smaller on process (nprocess-1).
           ref_p_array_rows is the size on all processes except process (nproces
s-1),
           rem_p_array_rows is the size on process (nprocess-1).
        */
```



```
        p_array_rows = (((array_rows / 8) + nprocess -1) / nprocess)*8;
        ref_p_array_rows = p_array_rows;
        rem_p_array_rows = p_array_rows - ((p_array_rows*nprocess) - array_rows)
;
        if(rank==(nprocess-1)){
            if((p_array_rows*(rank+1)) > array_rows) {
                p_array_rows = rem_p_array_rows;
            }
        }

        buildTestData();
    }

    public void JGFkernel() throws MPIException{

        Do();

    }

    public void JGFvalidate(){
        boolean error;

        if(rank==0) {
            error = false;
            for (int i = 0; i < array_rows; i++){
                error = (plain1 [i] != plain2 [i]);
                if (error){
                    System.out.println("Validation failed");
                    System.out.println("Original Byte " + i + "=" + plain1[i]);
                    System.out.println("Encrypted Byte " + i + "=" + crypt1[i]);
                    System.out.println("Decrypted Byte " + i + "=" + plain2[i]);
                    //break;
                }
            }
        }
    }

    public void JGFtidyup(){
        freeTestData();
    }

    public void JGFrun(int size) throws MPIException{

        JGFsetsize(size);
        JGFinitialise();
        JGFkernel();
        JGFvalidate();
        JGFtidyup();

    }

}
```

```
/**************************************************************************
 *                                                                        *
 *             Java Grande Forum Benchmark Suite - MPJ Version 1.0        *
 *                                                                        *
 *                                produced by                             *
 *                                                                        *
 *                    Java Grande Benchmarking Project                    *
 *                                                                        *
 *                                   at                                   *
 *                                                                        *
 *                  Edinburgh Parallel Computing Centre                   *
 *                                                                        *
 *                  email: epcc-javagrande@epcc.ed.ac.uk                  *
 *                                                                        *
 *                    Original version of this code by                    *
 *                    Gabriel Zachmann (zach@igd.fhg.de)                  *
 *                                                                        *
 *      This version copyright (c) The University of Edinburgh, 2001.     *
 *                          All rights reserved.                          *
 *                                                                        *
 **************************************************************************/


/**
 * Class IDEATest
 *
 * This test performs IDEA encryption then decryption. IDEA stands
 * for International Data Encryption Algorithm. The test is based
 * on code presented in Applied Cryptography by Bruce Schnier,
 * which was based on code developed by Xuejia Lai and James L.
 * Massey.

 **/

package uk.ac.warwick.dcs.hpsg.applications.jgf.crypt;

import java.util.*;
import mpi.*;

class IDEATest
{

    // Declare class data. Byte buffer plain1 holds the original
    // data for encryption, crypt1 holds the encrypted data, and
    // plain2 holds the decrypted data, which should match plain1
    // byte for byte.

    int array_rows;
    int p_array_rows;
    int ref_p_array_rows;
    int rem_p_array_rows;
```



```
    byte [] plain1 = null;       // Buffer for plaintext data.
    byte [] crypt1 = null;       // Buffer for encrypted data.
    byte [] plain2 = null;       // Buffer for decrypted data.

    byte [] p_plain1 = null;
    byte [] p_crypt1 = null;
    byte [] p_plain2 = null;

    short [] userkey;     // Key for encryption/decryption.
    int [] Z;             // Encryption subkey (userkey derived).
    int [] DK;            // Decryption subkey (userkey derived).

    void Do() throws MPIException
    {

        // temporary array for MPI
        int m_length;

        MPI.COMM_WORLD.Barrier();

        // broadcast information
        if(JGFCryptBench.rank==0) {
            for (int i = 0; i < p_array_rows; i++) {
                p_plain1[i] = plain1[i];
            }
            for(int k=1;k<JGFCryptBench.nprocess;k++){
                if(k==JGFCryptBench.nprocess-1) {
                    m_length = rem_p_array_rows;
                } else {
                    m_length = p_array_rows;
                }
                MPI.COMM_WORLD.Ssend(plain1,(p_array_rows*k),m_length,MPI.BYTE,k
,k);
            }
        } else {
            MPI.COMM_WORLD.Recv(p_plain1,0,p_array_rows,MPI.BYTE,0,JGFCryptBench
.rank);
        }

        MPI.COMM_WORLD.Barrier();

        cipher_idea(p_plain1, p_crypt1, Z);     // Encrypt plain1.
        cipher_idea(p_crypt1, p_plain2, DK);    // Decrypt.

        MPI.COMM_WORLD.Barrier();

        if(JGFCryptBench.rank==0) {
            for(int k=0; k<p_array_rows;k++){
                plain2[k] = p_plain2[k];
            }
```

```
            for(int k=1;k<JGFCryptBench.nprocess;k++) {
                MPI.COMM_WORLD.Recv(plain2,(p_array_rows*k),p_array_rows,MPI.BYT
E,k,k);
            }
        } else {
            MPI.COMM_WORLD.Ssend(p_plain2,0,p_array_rows,MPI.BYTE,0,JGFCryptBenc
h.rank);
        }

        MPI.COMM_WORLD.Barrier();

    }

    /*
     * buildTestData
     *
     * Builds the data used for the test -- each time the test is run.
     */

    void buildTestData()
    {

        // Create three byte arrays that will be used (and reused) for
        // encryption/decryption operations.

        if(JGFCryptBench.rank==0) {
            plain1 = new byte [array_rows];
            crypt1 = new byte [array_rows];
            plain2 = new byte [array_rows];
        }

        p_plain1 = new byte [p_array_rows];
        p_crypt1 = new byte [p_array_rows];
        p_plain2 = new byte [p_array_rows];

        Random rndnum = new Random(136506717L);  // Create random number generat
or.

        // Allocate three arrays to hold keys: userkey is the 128-bit key.
        // Z is the set of 16-bit encryption subkeys derived from userkey,
        // while DK is the set of 16-bit decryption subkeys also derived
        // from userkey. NOTE: The 16-bit values are stored here in
        // 32-bit int arrays so that the values may be used in calculations
        // as if they are unsigned. Each 64-bit block of plaintext goes
        // through eight processing rounds involving six of the subkeys
        // then a final output transform with four of the keys; (8 * 6)
        // + 4 = 52 subkeys.

        userkey = new short [8];  // User key has 8 16-bit shorts.
        Z = new int [52];         // Encryption subkey (user key derived).
        DK = new int [52];        // Decryption subkey (user key derived).
```



```
        // Generate user key randomly; eight 16-bit values in an array.

        for (int i = 0; i < 8; i++)
            {
                // Again, the random number function returns int. Converting
                // to a short type preserves the bit pattern in the lower 16
                // bits of the int and discards the rest.

                userkey[i] = (short) rndnum.nextInt();
            }

        // Compute encryption and decryption subkeys.

        calcEncryptKey();
        calcDecryptKey();

        // Fill plain1 with "text."
        // do on process 0 for reference

        if(JGFCryptBench.rank==0) {
            for (int i = 0; i < array_rows; i++)
                {
                    plain1[i] = (byte) i;

                    // Converting to a byte
                    // type preserves the bit pattern in the lower 8 bits of the
                    // int and discards the rest.
                }
        }

    }

    /*
     * calcEncryptKey
     *
     * Builds the 52 16-bit encryption subkeys Z[] from the user key and
     * stores in 32-bit int array. The routing corrects an error in the
     * source code in the Schnier book. Basically, the sense of the 7-
     * and 9-bit shifts are reversed. It still works reversed, but would
     * encrypted code would not decrypt with someone else's IDEA code.
     */

    private void calcEncryptKey()
    {
        int j;                          // Utility variable.

        for (int i = 0; i < 52; i++) // Zero out the 52-int Z array.
            Z[i] = 0;

        for (int i = 0; i < 8; i++)  // First 8 subkeys are userkey itself.
            {
                Z[i] = userkey[i] & 0xffff;     // Convert "unsigned"
```

```
            // short to int.
        }

    // Each set of 8 subkeys thereafter is derived from left rotating
    // the whole 128-bit key 25 bits to left (once between each set of
    // eight keys and then before the last four). Instead of actually
    // rotating the whole key, this routine just grabs the 16 bits
    // that are 25 bits to the right of the corresponding subkey
    // eight positions below the current subkey. That 16-bit extent
    // straddles two array members, so bits are shifted left in one
    // member and right (with zero fill) in the other. For the last
    // two subkeys in any group of eight, those 16 bits start to
    // wrap around to the first two members of the previous eight.

    for (int i = 8; i < 52; i++)
        {
            j = i % 8;
            if (j < 6)
                {
                    Z[i] = ((Z[i -7]>>>9) | (Z[i-6]<<7)) // Shift and combin
e.
                        & 0xFFFF;                   // Just 16 bits.
                    continue;                           // Next iteration.
                }

            if (j == 6)     // Wrap to beginning for second chunk.
                {
                    Z[i] = ((Z[i -7]>>>9) | (Z[i-14]<<7))
                        & 0xFFFF;
                    continue;
                }

            // j == 7 so wrap to beginning for both chunks.

            Z[i] = ((Z[i -15]>>>9) | (Z[i-14]<<7))
                & 0xFFFF;
        }
}

/*
 * calcDecryptKey
 *
 * Builds the 52 16-bit encryption subkeys DK[] from the encryption-
 * subkeys Z[]. DK[] is a 32-bit int array holding 16-bit values as
 * unsigned.
 */

private void calcDecryptKey()
{
    int j, k;                   // Index counters.
    int t1, t2, t3;             // Temps to hold decrypt subkeys.
```



```
    t1 = inv(Z[0]);             // Multiplicative inverse (mod x10001).
    t2 = - Z[1] & 0xffff;       // Additive inverse, 2nd encrypt subkey.
    t3 = - Z[2] & 0xffff;       // Additive inverse, 3rd encrypt subkey.

    DK[51] = inv(Z[3]);         // Multiplicative inverse (mod x10001).
    DK[50] = t3;
    DK[49] = t2;
    DK[48] = t1;

    j = 47;                     // Indices into temp and encrypt arrays.
    k = 4;
    for (int i = 0; i < 7; i++)
        {
            t1 = Z[k++];
            DK[j--] = Z[k++];
            DK[j--] = t1;
            t1 = inv(Z[k++]);
            t2 = -Z[k++] & 0xffff;
            t3 = -Z[k++] & 0xffff;
            DK[j--] = inv(Z[k++]);
            DK[j--] = t2;
            DK[j--] = t3;
            DK[j--] = t1;
        }

    t1 = Z[k++];
    DK[j--] = Z[k++];
    DK[j--] = t1;
    t1 = inv(Z[k++]);
    t2 = -Z[k++] & 0xffff;
    t3 = -Z[k++] & 0xffff;
    DK[j--] = inv(Z[k++]);
    DK[j--] = t3;
    DK[j--] = t2;
    DK[j--] = t1;
}

/*
 * cipher_idea
 *
 * IDEA encryption/decryption algorithm. It processes plaintext in
 * 64-bit blocks, one at a time, breaking the block into four 16-bit
 * unsigned subblocks. It goes through eight rounds of processing
 * using 6 new subkeys each time, plus four for last step. The source
 * text is in array text1, the destination text goes into array text2
 * The routine represents 16-bit subblocks and subkeys as type int so
 * that they can be treated more easily as unsigned. Multiplication
 * modulo 0x10001 interprets a zero sub-block as 0x10000; it must to
 * fit in 16 bits.
 */

private void cipher_idea(byte [] text1, byte [] text2, int [] key)
```

```
    {

        int i1 = 0;                 // Index into first text array.
        int i2 = 0;                 // Index into second text array.
        int ik;                     // Index into key array.
        int x1, x2, x3, x4, t1, t2; // Four "16-bit" blocks, two temps.
        int r;                      // Eight rounds of processing.

        for (int i = 0; i < text1.length; i += 8)
            {

                ik = 0;                 // Restart key index.
                r = 8;                  // Eight rounds of processing.

                // Load eight plain1 bytes as four 16-bit "unsigned" integers.
                // Masking with 0xff prevents sign extension with cast to int.

                x1 = text1[i1++] & 0xff;          // Build 16-bit x1 from 2 byte
s,
                x1 |= (text1[i1++] & 0xff) << 8;  // assuming low-order byte fir
st.
                x2 = text1[i1++] & 0xff;
                x2 |= (text1[i1++] & 0xff) << 8;
                x3 = text1[i1++] & 0xff;
                x3 |= (text1[i1++] & 0xff) << 8;
                x4 = text1[i1++] & 0xff;
                x4 |= (text1[i1++] & 0xff) << 8;

                do {
                    // 1) Multiply (modulo 0x10001), 1st text sub-block
                    // with 1st key sub-block.

                    x1 = (int) ((long) x1 * key[ik++] % 0x10001L & 0xffff);

                    // 2) Add (modulo 0x10000), 2nd text sub-block
                    // with 2nd key sub-block.

                    x2 = x2 + key[ik++] & 0xffff;

                    // 3) Add (modulo 0x10000), 3rd text sub-block
                    // with 3rd key sub-block.

                    x3 = x3 + key[ik++] & 0xffff;

                    // 4) Multiply (modulo 0x10001), 4th text sub-block
                    // with 4th key sub-block.

                    x4 = (int) ((long) x4 * key[ik++] % 0x10001L & 0xffff);

                    // 5) XOR results from steps 1 and 3.

                    t2 = x1 ^ x3;
```



```
                    // 6) XOR results from steps 2 and 4.
                    // Included in step 8.

                    // 7) Multiply (modulo 0x10001), result of step 5
                    // with 5th key sub-block.

                    t2 = (int) ((long) t2 * key[ik++] % 0x10001L & 0xffff);

                    // 8) Add (modulo 0x10000), results of steps 6 and 7.

                    t1 = t2 + (x2 ^ x4) & 0xffff;

                    // 9) Multiply (modulo 0x10001), result of step 8
                    // with 6th key sub-block.

                    t1 = (int) ((long) t1 * key[ik++] % 0x10001L & 0xffff);

                    // 10) Add (modulo 0x10000), results of steps 7 and 9.

                    t2 = t1 + t2 & 0xffff;

                    // 11) XOR results from steps 1 and 9.

                    x1 ^= t1;

                    // 14) XOR results from steps 4 and 10. (Out of order).

                    x4 ^= t2;

                    // 13) XOR results from steps 2 and 10. (Out of order).

                    t2 ^= x2;

                    // 12) XOR results from steps 3 and 9. (Out of order).

                    x2 = x3 ^ t1;

                    x3 = t2;        // Results of x2 and x3 now swapped.

                } while(--r != 0);  // Repeats seven more rounds.

                // Final output transform (4 steps).

                // 1) Multiply (modulo 0x10001), 1st text-block
                // with 1st key sub-block.

                x1 = (int) ((long) x1 * key[ik++] % 0x10001L & 0xffff);

                // 2) Add (modulo 0x10000), 2nd text sub-block
                // with 2nd key sub-block. It says x3, but that is to undo swap
                // of subblocks 2 and 3 in 8th processing round.
```

264

```
                x3 = x3 + key[ik++] & 0xffff;

                // 3) Add (modulo 0x10000), 3rd text sub-block
                // with 3rd key sub-block. It says x2, but that is to undo swap
                // of subblocks 2 and 3 in 8th processing round.

                x2 = x2 + key[ik++] & 0xffff;

                // 4) Multiply (modulo 0x10001), 4th text-block
                // with 4th key sub-block.

                x4 = (int) ((long) x4 * key[ik++] % 0x10001L & 0xffff);

                // Repackage from 16-bit sub-blocks to 8-bit byte array text2.

                text2[i2++] = (byte) x1;
                text2[i2++] = (byte) (x1 >>> 8);
                text2[i2++] = (byte) x3;                    // x3 and x2 are switche
d
                text2[i2++] = (byte) (x3 >>> 8);            // only in name.
                text2[i2++] = (byte) x2;
                text2[i2++] = (byte) (x2 >>> 8);
                text2[i2++] = (byte) x4;
                text2[i2++] = (byte) (x4 >>> 8);

            }   // End for loop.

        }   // End routine.

        /*
         * mul
         *
         * Performs multiplication, modulo (2**16)+1. This code is structured
         * on the assumption that untaken branches are cheaper than taken
         * branches, and that the compiler doesn't schedule branches.
         * Java: Must work with 32-bit int and one 64-bit long to keep
         * 16-bit values and their products "unsigned." The routine assumes
         * that both a and b could fit in 16 bits even though they come in
         * as 32-bit ints. Lots of "& 0xFFFF" masks here to keep things 16-bit.
         * Also, because the routine stores mod (2**16)+1 results in a 2**16
         * space, the result is truncated to zero whenever the result would
         * zero, be 2**16. And if one of the multiplicands is 0, the result
         * is not zero, but (2**16) + 1 minus the other multiplicand (sort
         * of an additive inverse mod 0x10001).

         * NOTE: The java conversion of this routine works correctly, but
         * is half the speed of using Java's modulus division function (%)
         * on the multiplication with a 16-bit masking of the result--running
         * in the Symantec Caje IDE. So it's not called for now; the test
         * uses Java % instead.
         */
```



```
    private int mul(int a, int b) throws ArithmeticException
    {
        long p;             // Large enough to catch 16-bit multiply
        // without hitting sign bit.
        if (a != 0)
            {
                if(b != 0)
                    {
                        p = (long) a * b;
                        b = (int) p & 0xFFFF;       // Lower 16 bits.
                        a = (int) p >>> 16;         // Upper 16 bits.

                        return (b - a + (b < a ? 1 : 0) & 0xFFFF);
                    }
                else
                    return ((1 - a) & 0xFFFF);  // If b = 0, then same as
                // 0x10001 - a.
            }
        else                                    // If a = 0, then return
            return((1 - b) & 0xFFFF);           // same as 0x10001 - b.
    }

    /*
     * inv
     *
     * Compute multiplicative inverse of x, modulo (2**16)+1 using
     * extended Euclid's GCD (greatest common divisor) algorithm.
     * It is unrolled twice to avoid swapping the meaning of
     * the registers. And some subtracts are changed to adds.
     * Java: Though it uses signed 32-bit ints, the interpretation
     * of the bits within is strictly unsigned 16-bit.
     */

    private int inv(int x)
    {
        int t0, t1;
        int q, y;

        if (x <= 1)                 // Assumes positive x.
            return(x);              // 0 and 1 are self-inverse.

        t1 = 0x10001 / x;           // (2**16+1)/x; x is >= 2, so fits 16 bits.
        y = 0x10001 % x;
        if (y == 1)
            return((1 - t1) & 0xFFFF);

        t0 = 1;
        do {
            q = x / y;
            x = x % y;
            t0 += q * t1;
```

```
            if (x == 1) return(t0);
            q = y / x;
            y = y % x;
            t1 += q * t0;
        } while (y != 1);

        return((1 - t1) & 0xFFFF);
    }

    /*
     * freeTestData
     *
     * Nulls arrays and forces garbage collection to free up memory.
     */

    void freeTestData()
    {
        plain1 = null;
        crypt1 = null;
        plain2 = null;
        p_plain1 = null;
        p_crypt1 = null;
        p_plain2 = null;
        userkey = null;
        Z = null;
        DK = null;

        System.gc();                // Force garbage collection.
    }

}
```

```
/*
 *  Copyright (C) 1998, University of Adelaide, under its participation
 *  in the Advanced Computational Systems Cooperative Research Centre
 *  Agreement.
 *
 *  THIS SOFTWARE IS MADE AVAILABLE, AS IS, AND THE UNIVERSITY
 *  OF ADELAIDE DOES NOT MAKE ANY WARRANTY ABOUT THE SOFTWARE, ITS
 *  PERFORMANCE, ITS MERCHANTABILITY OR FITNESS FOR ANY PARTICULAR
 *  USE, FREEDOM FROM ANY COMPUTER DISEASES OR ITS CONFORMITY TO ANY
 *  SPECIFICATION. THE ENTIRE RISK AS TO QUALITY AND PERFORMANCE OF
 *  THE SOFTWARE IS WITH THE USER.
 *
 *  Copyright of the software and supporting documentation is owned by the
 *  University of Adelaide, and free access is hereby granted as a license
 *  to use this software, copy this software and prepare derivative works
 *  based upon this software.  However, any distribution of this software
 *  source code or supporting documentation or derivative works (source
 *  code and supporting documentation) must include this copyright notice
 *  and acknowledge the University of Adelaide.
 *
 *
 *  Developed by: Distributed High Performance Computing (DHPC) Group
 *                Department of Computer Science
 *                The University of Adelaide
 *                South Australia 5005
 *                Tel +61 8 8303 4519, Fax +61 8 8303 4366
 *                http://dhpc.adelaide.edu.au
 *  Last Modified: 26 January 1999
 *
 *
 ********************************************************************************
 *                      Java Benchmarks
 *
 *  These benchmarks complement the Java Grande Benchmarks and utilise
 *  the timing and result reporting framework provided with them.
 *
 *  Section1a includes extra low level benchmarks, Section2a includes extra
 *  kernel benchmarks and util includes commonly used code such as for
 *  printing header information
 *
 *  Jesudas Mathew (jm@cs.adelaide.edu.au)
 ********************************************************************************
 *
 *  NAS EP Benchmark
 */

package uk.ac.warwick.dcs.hpsg.applications.dhpc.ep;

public class DHPC_EPBench extends KernelEP {

    //  private int size;
```



```
    //  private int datasizes[]={1,2,3};  // Only a single size (Class S) is sup
ported

    public void JGFsetsize(int size) {

        this.size = size;

        /* This is my doing to allow the kernel to be parameterised for a power
of 2 */
        n = (long)Math.pow(2, size);
    }

    public void JGFinitialise() {}

    public void JGFkernel() {
        ep();
    }

    public void JGFvalidate() {}

    public void JGFtidyup() {}

    public void JGFrun(int size) {

        JGFsetsize(size);
        JGFinitialise();
        JGFkernel();
        JGFvalidate();
        JGFtidyup();

    }

    public static void main(String argv[]) {

        if (argv.length != 1)
            System.err.println("Usage: java DHPC_EPBench Datasize (any positive integer - original
value = 24)");

        else {

            int size = new Integer(argv[0]).intValue();

            DHPC_EPBench eb = new DHPC_EPBench();
            eb.JGFrun(size);

        }
    }
}
```

```
/*
 *  Copyright (C) 1998, University of Adelaide, under its participation
 *  in the Advanced Computational Systems Cooperative Research Centre
 *  Agreement.
 *
 *  THIS SOFTWARE IS MADE AVAILABLE, AS IS, AND THE UNIVERSITY
 *  OF ADELAIDE DOES NOT MAKE ANY WARRANTY ABOUT THE SOFTWARE, ITS
 *  PERFORMANCE, ITS MERCHANTABILITY OR FITNESS FOR ANY PARTICULAR
 *  USE, FREEDOM FROM ANY COMPUTER DISEASES OR ITS CONFORMITY TO ANY
 *  SPECIFICATION. THE ENTIRE RISK AS TO QUALITY AND PERFORMANCE OF
 *  THE SOFTWARE IS WITH THE USER.
 *
 *  Copyright of the software and supporting documentation is owned by the
 *  University of Adelaide, and free access is hereby granted as a license
 *  to use this software, copy this software and prepare derivative works
 *  based upon this software.  However, any distribution of this software
 *  source code or supporting documentation or derivative works (source
 *  code and supporting documentation) must include this copyright notice
 *  and acknowledge the University of Adelaide.
 *
 *
 *  Developed by: Distributed High Performance Computing (DHPC) Group
 *                Department of Computer Science
 *                The University of Adelaide
 *                South Australia 5005
 *                Tel +61 8 8303 4519, Fax +61 8 8303 4366
 *                http://dhpc.adelaide.edu.au
 *  Last Modified: 26 January 1999
 *
 *
 ****************************************************************************
 *                      Java Benchmarks
 *
 *  These benchmarks complement the Java Grande Benchmarks and utilise
 *  the timing and result reporting framework provided with them.
 *
 *  Section1a includes extra low level benchmarks, Section2a includes extra
 *  kernel benchmarks and util includes commonly used code such as for
 *  printing header information
 *
 *  Jesudas Mathew (jm@cs.adelaide.edu.au)
 ****************************************************************************
 *
 *  This class implements the NAS Embarrisingly Parallel Benchmark
 *  (Class S)
 */

package uk.ac.warwick.dcs.hpsg.applications.dhpc.ep;

public class KernelEP {

    protected int size;
```



```
    protected long n; // n = 2^size
    //  protected int[] datasizes = { 8388608, 16777216, 33554432 };

    //static final int n=16777216;  // Number of random numbers to generate
    //  static final int s=271828183; // Inital seed

    public void ep() {
        // Generate Gaussian random number pairs in accordance
        // with NAS specifications

        double x,y,t,xy;
        int l;
        double xx=0.0f;
        double yy=0.0f;
        int Q[] = new int[10];
        double Xsum=0.0f;
        double Ysum=0.0f;

        for (int i=0;i<10;i++) Q[i]=0;

        for (int j=0;j<n;j++) {
            x = 2.0*next()-1.0; y=2.0*next()-1.0;
            t=x*x+y*y;
            if (t<=1.0) {
                xy= (double) Math.sqrt((-2.0*Math.log(t))/t);
                xx = x*xy; Xsum+=xx;
                yy = y*xy; Ysum+=yy;
                l = (int) Math.floor(Math.max(Math.abs(xx),Math.abs(yy)));
                Q[l]++;
            }
        }

        /* Get rid of any print out statements from kernel */
        //              System.out.println("Sum X:"+Xsum);
        //              System.out.println("Sum Y:"+Ysum);
        //              for (l=0;l<10;l++)
        //                      System.out.println("Q["+l+"]:"+Q[l]);

    }

    // Implements a 64-bit linear congruential generator of the form
    // x[n+1]=A*x[n] mod 2**46
    // Code derived from serial NAS reference C code

    static double X=271828183.0;  // Inital seed
    static final double A=1220703125.0; // Multiplier
    static int KS;
    static double       R23, R46, T23, T46;

    double next()
    {
        double          T1, T2, T3, T4;
```

```
        double          A1;
        double          A2;
        double          X1;
        double          X2;
        double          Z;
        int                     i, j;

        if (KS == 0)
            {
                R23 = 1.0;
                R46 = 1.0;
                T23 = 1.0;
                T46 = 1.0;

                for (i=1; i<=23; i++)
                    {
                        R23 = 0.50 * R23;
                        T23 = 2.0 * T23;
                    }
                for (i=1; i<=46; i++)
                    {
                        R46 = 0.50 * R46;
                        T46 = 2.0 * T46;
                    }
                KS = 1;
            }

        /*  Break A into two parts such that A = 2^23 * A1 + A2 and set X = N.
*/

        T1 = R23 * A;
        j  = (int) T1;
        A1 = j;
        A2 = A - T23 * A1;

        /*  Break X into two parts such that X = 2^23 * X1 + X2, compute
            Z = A1 * X2 + A2 * X1  (mod 2^23), and then
            X = 2^23 * Z + A2 * X2  (mod 2^46).                          */

        T1 = R23 * X;
        j  = (int) T1;
        X1 = j;
        X2 = X - T23 * X1;
        T1 = A1 * X2 + A2 * X1;

        j  = (int) (R23 * T1);
        T2 = j;
        Z = T1 - T23 * T2;
        T3 = T23 * Z + A2 * X2;
        j  = (int) (R46 * T3);
        T4 = j;
        X = T3 - T46 * T4;
```



```
        return(R46 * X);
    }
}
```

```
/*
 *  Copyright (C) 1998, University of Adelaide, under its participation
 *  in the Advanced Computational Systems Cooperative Research Centre
 *  Agreement.
 *
 *  THIS SOFTWARE IS MADE AVAILABLE, AS IS, AND THE UNIVERSITY
 *  OF ADELAIDE DOES NOT MAKE ANY WARRANTY ABOUT THE SOFTWARE, ITS
 *  PERFORMANCE, ITS MERCHANTABILITY OR FITNESS FOR ANY PARTICULAR
 *  USE, FREEDOM FROM ANY COMPUTER DISEASES OR ITS CONFORMITY TO ANY
 *  SPECIFICATION. THE ENTIRE RISK AS TO QUALITY AND PERFORMANCE OF
 *  THE SOFTWARE IS WITH THE USER.
 *
 *  Copyright of the software and supporting documentation is owned by the
 *  University of Adelaide, and free access is hereby granted as a license
 *  to use this software, copy this software and prepare derivative works
 *  based upon this software.  However, any distribution of this software
 *  source code or supporting documentation or derivative works (source
 *  code and supporting documentation) must include this copyright notice
 *  and acknowledge the University of Adelaide.
 *
 *
 *  Developed by: Distributed High Performance Computing (DHPC) Group
 *                Department of Computer Science
 *                The University of Adelaide
 *                South Australia 5005
 *                Tel +61 8 8303 4519, Fax +61 8 8303 4366
 *                http://dhpc.adelaide.edu.au
 *  Last Modified: 26 January 1999
 *
 *
 ************************************************************************
 *                      Java Benchmarks
 *
 *  These benchmarks complement the Java Grande Benchmarks and utilise
 *  the timing and result reporting framework provided with them.
 *
 *  Section1a includes extra low level benchmarks, Section2a includes extra
 *  kernel benchmarks and util includes commonly used code such as for
 *  printing header information
 *
 *  Jesudas Mathew (jm@cs.adelaide.edu.au)
 ************************************************************************
 *
 *  256 Point Fast Fourier Transform and Inverse Transform
 *  Derived from source provided by Ken Hawick
 */

package uk.ac.warwick.dcs.hpsg.applications.dhpc.fft;

public class DHPC_FFTBench {

    private int size;
```



```
    static int n1=64;
    static int n2=64;
    static int n3=64;
    static int iterations=6;
    static double data[];
    static double data2[];

    public void JGFsetsize(int size) {
        this.size = size;
    }

    public void JGFinitialise() {
        init();
    }

    public void JGFkernel() {
        doFFT();
    }

    public void JGFvalidate() {}

    public void JGFtidyup() {}

    public void JGFrun(int size){

        JGFsetsize(size);
        JGFinitialise();
        JGFkernel();
        JGFvalidate();
        JGFtidyup();

    }

    public DHPC_FFTBench() {}

    public static void main (String[] argv) {

        if (argv.length != 3)
            System.err.println("Usage: java DHPC_FFTBench n1 n2 n3 (a positive integer - original v
alue = 64)");

        else {

            n1 = new Integer(argv[0]).intValue();
            n2 = new Integer(argv[1]).intValue();
            n3 = new Integer(argv[2]).intValue();

            DHPC_FFTBench fftBench = new DHPC_FFTBench();
            fftBench.JGFrun(0);

        }
    }
```

```
public void doFFT() {

    int nn3 = n1*n2*n3*2;
    int nn[] = new int[3];

    data = new double[nn3];
    data2= new double[nn3];

    nn[0] = n1;
    nn[1] = n2;
    nn[2] = n3;

    int i;
    for(i=0;i<nn3;i++) {
        data[i] = next();
    }

    complex_fourierndd(data, nn);

    for (i=1;i<=6;i++) {
        evolve(data,nn,i);
        inverse_complex_fourierndd(data2, nn);
        double X1=0.0, X2=0.0;
        int q,r,s,pos;
        for (int j=0;j<1024;j++) {
            q = j % n1;
            r = (3*j) % n2;
            s = (5*j) % n3;
            pos = 2*(n1*n2*s+n1*r+q);
            X1+= data[pos];
            X2+= data[pos+1];
        }

    }

}

public static void inter(double data[]) {
    double c1,c2;
    int pos;

    //int j=13;int k=7;
    for (int k=0; k<n3; k++) {
        c1=0.0; c2=0.0;
        for (int j=0; j<n2; j++)
            for (int i=0; i<n1; i++) {
                pos=2*(k*n1*n2+j*n1+i);
                c1+=data[pos]; c2+=data[pos+1];
                System.out.println(data[pos]+","+data[pos+1]);
            }
        //  System.out.println(c1+","+c2);
```



```
        }
    }

    public static void evolve(double data[], int nn[],int t) {
        int pos=0;

        double pipi= Math.PI*Math.PI;
        double a = -4.0*t*pipi/1000000.0;
        for (int i=0;i<n1;i++)
            for (int j=0;j<n2;j++)
                for (int k=0;k<n3;k++) {
                    pos = 2*(k*n1*n2+j*n1+i);
                    data2[pos]=Math.exp(a*map(i,j,k))*data[pos];
                    data2[pos+1]=Math.exp(a*map(i,j,k))*data[pos+1];
                }
    }

    public static double map(int i,int j,int k) {
        int ii,jj,kk;
        ii= i < n1/2 ? i : i-n1;
        jj= j < n2/2 ? j : j-n2;
        kk= k < n3/2 ? k : k-n3;
        return((double) (ii*ii+jj*jj+kk*kk));
    }

    public static void complex_fourierndd( double data[], int nn[] ){
        auxiliary_complex_fourierndd( data, nn, 1);
    }

    public static void inverse_complex_fourierndd( double data[], int nn[] ){
        int i;
        double scale;
        auxiliary_complex_fourierndd( data, nn, -1);
        scale = data.length / 2;
        for(i=0;i<data.length;i++){
            data[i] /= scale;
        }

    }

    public static void auxiliary_complex_fourierndd(double data[], int nn[], int
isign)
    {
        int idim;
        int i1,i2,i3,i2rev,i3rev,ip1,ip2,ip3,ifp1,ifp2;
        int ibit,k1,k2,n,nprev,nrem,ntot;
        double tempi,tempr;
        double theta,wi,wpi,wpr,wr,wtemp;
        double wswap;

        int ndim = nn.length;
```

```
        // need to check all the dimensions are correct powers of two and consis
tent

        for (ntot=1,idim=0;idim<ndim;idim++)
            ntot *= nn[idim];
        nprev=1;
        for (idim=ndim;idim>=1;idim--) {
            n = nn[idim-1];
            nrem = ntot / ( n * nprev );
            ip1 = nprev << 1;
            ip2 = ip1 * n;
            ip3 = ip2 * nrem;
            i2rev = 1;
            for (i2=1;i2<=ip2;i2+=ip1) {

                if (i2 < i2rev) {

                    for (i1=i2;i1<=i2+ip1-2;i1+=2) {

                        for (i3=i1;i3<=ip3;i3+=ip2) {

                            i3rev = i2rev + i3 - i2;
                            wswap = data[i3-1]; data[i3-1] = data[i3rev-1]; data
[i3rev-1] = wswap;

                            wswap = data[i3];   data[i3]   = data[i3rev];   data
[i3rev]   = wswap;
                        }

                    }

                }

                ibit = ip2 >>> 1;

                while (ibit >= ip1 && i2rev > ibit) {

                    i2rev -= ibit;
                    ibit >>>= 1;
                }
                i2rev += ibit;
            }
            ifp1 = ip1;
            while (ifp1 < ip2) {
                ifp2 = ifp1 << 1;
                theta = isign * java.lang.Math.PI / ( ifp2/ip1 );
                wtemp = java.lang.Math.sin( 0.5 * theta );
                wpr = -2.0 * wtemp * wtemp;
                wpi = java.lang.Math.sin(theta);
                wr = 1.0;
                wi = 0.0;

                for (i3=1;i3<=ifp1;i3+=ip1) {
```



```
                    for (i1=i3;i1<=i3+ip1-2;i1+=2) {
                        for (i2=i1;i2<=ip3;i2+=ifp2) {
                            k1 = i2;
                            k2 = k1 + ifp1;
                            tempr = wr * data[k2-1] - wi * data[k2];
                            tempi = wr * data[k2]   + wi * data[k2-1];
                            data[k2-1] = data[k1-1] - tempr;
                            data[k2]   = data[k1]   - tempi;
                            data[k1-1] += tempr;
                            data[k1]   += tempi;
                        }
                    }
                    wtemp = wr;
                    wr = wr * wpr - wi * wpi + wr;
                    wi = wi * wpr + wtemp * wpi + wi;
                }

                ifp1=ifp2;
            }
            nprev *= n;
        }
    }

    static double X=314159265.0;  // Inital seed
    static final double A=1220703125.0; // Multiplier
    static double       R23, R46, T23, T46;

    void init() {
        int i;
        R23 = 1.0;
        R46 = 1.0;
        T23 = 1.0;
        T46 = 1.0;

        for (i=1; i<=23; i++)
            {
                R23 = 0.50 * R23;
                T23 = 2.0 * T23;
            }
        for (i=1; i<=46; i++)
            {
                R46 = 0.50 * R46;
                T46 = 2.0 * T46;
            }
    }

    double next()
    {
        X = mult(A,X);
        return(R46 * X);
    }
```

```
    double mult(double A, double X) {
        double          T1, T2, T3, T4;
        double          A1;
        double          A2;
        double          X1;
        double          X2;
        double          Z;
        int                     j;
        /*  Break A into two parts such that A = 2^23 * A1 + A2 and set X = N.
*/


        T1 = R23 * A;
        j  = (int) T1;
        A1 = j;
        A2 = A - T23 * A1;

        /*  Break X into two parts such that X = 2^23 * X1 + X2, compute
            Z = A1 * X2 + A2 * X1  (mod 2^23), and then
            X = 2^23 * Z + A2 * X2  (mod 2^46).                          */

        T1 = R23 * X;
        j  = (int) T1;
        X1 = j;
        X2 = X - T23 * X1;
        T1 = A1 * X2 + A2 * X1;

        j  = (int) (R23 * T1);
        T2 = j;
        Z = T1 - T23 * T2;
        T3 = T23 * Z + A2 * X2;
        j  = (int) (R46 * T3);
        T4 = j;
        X = T3 - T46 * T4;

        return(X);
    }
}
```

272

Appendix B

JavaGrande Benchmark Performance Characterisations

The following twenty-five pages contain the complete performance characterisations of the five JavaGrande benchmarks as documented in this thesis. The Sparse Matrix Multiply characterisation contains the '`smm.app.xml`', '`smm.tranmap`', '`smm-init.tran.xml`' and '`smm-kernel.tran.xml`' performance objects. The Fourier Coefficient Analysis characterisation contains the '`series.app.xml`', '`series.tranmap.xml`' and '`series-kernel.tran.xml`' performance objects. The IDEA Encryption characterisation contains the '`crypt.app.xml`', '`crypt.tranmap.xml`', '`crypt-init.tran.xml`', '`crypt-encrypt.tran.xml`' and '`crypt-finalise.tran.xml`' performance objects. The Gaussian Random Number Generation characterisation contains the '`ep.app.xml`', '`ep.tranmap.xml`' and '`ep-kernel.tran.xml`' performance objects. The Fast Fourier Transform characterisation contains the '`fft.app.xml`', '`fft.tranmap.xml`' and '`fft-kernel.tran.xml`' performance objects.

```
<?xml version="1.0" encoding="UTF-8"?>

<jPACE:application>

  <jPACE:confidence max="1"/>

  <jPACE:platform resource="mscs-01.dcs.warwick.ac.uk" vm="sun-hotspot-i386-linu
x-1.4.1_01"/>
  <jPACE:platform resource="mscs-02.dcs.warwick.ac.uk" vm="sun-hotspot-i386-linu
x-1.4.1_01"/>
  <jPACE:platform resource="mscs-03.dcs.warwick.ac.uk" vm="sun-hotspot-i386-linu
x-1.4.1_01"/>
  <jPACE:platform resource="mscs-04.dcs.warwick.ac.uk" vm="sun-hotspot-i386-linu
x-1.4.1_01"/>
  <jPACE:platform resource="mscs-05.dcs.warwick.ac.uk" vm="sun-hotspot-i386-linu
x-1.4.1_01"/>
  <jPACE:platform resource="mscs-06.dcs.warwick.ac.uk" vm="sun-hotspot-i386-linu
x-1.4.1_01"/>
  <jPACE:platform resource="mscs-07.dcs.warwick.ac.uk" vm="sun-hotspot-i386-linu
x-1.4.1_01"/>
  <jPACE:platform resource="mscs-08.dcs.warwick.ac.uk" vm="sun-hotspot-i386-linu
x-1.4.1_01"/>
  <jPACE:platform resource="mscs-09.dcs.warwick.ac.uk" vm="sun-hotspot-i386-linu
x-1.4.1_01"/>
  <jPACE:platform resource="mscs-10.dcs.warwick.ac.uk" vm="sun-hotspot-i386-linu
x-1.4.1_01"/>
  <jPACE:platform resource="mscs-11.dcs.warwick.ac.uk" vm="sun-hotspot-i386-linu
x-1.4.1_01"/>
  <jPACE:platform resource="mscs-12.dcs.warwick.ac.uk" vm="sun-hotspot-i386-linu
x-1.4.1_01"/>
  <jPACE:platform resource="mscs-13.dcs.warwick.ac.uk" vm="sun-hotspot-i386-linu
x-1.4.1_01"/>
  <jPACE:platform resource="mscs-14.dcs.warwick.ac.uk" vm="sun-hotspot-i386-linu
x-1.4.1_01"/>
  <jPACE:platform resource="mscs-15.dcs.warwick.ac.uk" vm="sun-hotspot-i386-linu
x-1.4.1_01"/>
  <jPACE:platform resource="mscs-16.dcs.warwick.ac.uk" vm="sun-hotspot-i386-linu
x-1.4.1_01"/>

  <jPACE:parameter name="M" value="500000"/>
  <jPACE:parameter name="N" value="500000"/>
  <jPACE:parameter name="nz" value="2500000"/>
  <jPACE:parameter name="SPARSE_NUM_ITER" value="200"/>

  <jPACE:link targetObject="smm.tranmap" targetVariable="M" newValue="${M}"/>
  <jPACE:link targetObject="smm.tranmap" targetVariable="N" newValue="${N}"/>
  <jPACE:link targetObject="smm.tranmap" targetVariable="nz" newValue="${nz}"/>
  <jPACE:link targetObject="smm.tranmap" targetVariable="SPARSE_NUM_ITER" newVal
ue="${SPARSE_NUM_ITER}"/>

  <jPACE:proc name="main">
    <jPACE:evaluateTransactionMap transactionMap="smm.tranmap" platforms="ALL"/>
```



```
  </jPACE:proc>

</jPACE:application>
```

```
<?xml version="1.0" encoding="UTF-8"?>

<jPACE:transactionMap>

  <jPACE:variable name="M"/>
  <jPACE:variable name="N"/>
  <jPACE:variable name="nz"/>
  <jPACE:variable name="SPARSE_NUM_ITER"/>

  <jPACE:link targetObject="smm-init.tran" targetVariable="N" newValue="${N}"/>
  <jPACE:link targetObject="smm-init.tran" targetVariable="nz" newValue="${nz}"/
>
  <jPACE:link targetObject="smm-kernel.tran" targetVariable="M" newValue="${M}"/
>
  <jPACE:link targetObject="smm-kernel.tran" targetVariable="nz" newValue="${nz}
"/>
  <jPACE:link targetObject="smm-kernel.tran" targetVariable="SPARSE_NUM_ITER" ne
wValue="${SPARSE_NUM_ITER}"/>

  <jPACE:proc name="main">
    <jPACE:evaluateMap map="smm.map"/>
  </jPACE:proc>

  <jPACE:map name="smm.map">

    <jPACE:step>
      <jPACE:evaluateTransaction transaction="smm-init.tran" platforms="ALL"/>
    </jPACE:step>

    <jPACE:step>
      <jPACE:evaluateTransaction transaction="smm-kernel.tran" platforms="ALL"/>
    </jPACE:step>

  </jPACE:map>

</jPACE:transactionMap>
```

```
<?xml version="1.0" encoding="UTF-8"?>

<jPACE:transaction>

  <jPACE:variable name="N"/>
  <jPACE:variable name="nz"/>
  <jPACE:variable name="p_datasizes_nz"/>

  <jPACE:confidence variable="c">
    <jPACE:setVariable variable="c" value="1"/>
  </jPACE:confidence>

  <jPACE:proc name="main">
    <jPACE:setVariable variable="p_datasizes_nz" value="(${nz} + ${nP} - 1) / ${
nP}"/>

    <jPACE:if leftExpression="${cP}" condition="EQUALS" rightExpression="${nP} -
 1">
      <jPACE:if leftExpression="${p_datasizes_nz}*${nP}" condition="GREATER_THAN
" rightExpression="${nz}">
        <jPACE:setVariable variable="p_datasizes_nz" value="${p_datasizes_nz} -
((${p_datasizes_nz}*${nP}) - ${nz})"/>
      </jPACE:if>
    </jPACE:if>

    <jPACE:evaluateMethod class="uk.ac.warwick.dcs.hpsg.applications.jgf.sparsem
atmult.JGFSparseMatmultBench"
                          method="JGFinitialise" descriptor="()V"/>
  </jPACE:proc>

  <jPACE:method class="java.lang.Math"
                method="abs" descriptor="(I)I" type="transaction">
  </jPACE:method>

  <jPACE:method class="java.util.Random"
                method="nextInt" descriptor="()I" type="transaction">
  </jPACE:method>

  <jPACE:method class="java.util.Random"
                method="nextDouble" descriptor="()D" type="transaction">
  </jPACE:method>

  <jPACE:method class="uk.ac.warwick.dcs.hpsg.applications.jgf.sparsematmult.JGF
SparseMatmultBench"
                method="RandomVector" descriptor="()[D" type="characterised">

    <jPACE:bytecodeBlock id="RandomVector()[D:1"/>

    <jPACE:loop count="${N}">

      <jPACE:bytecodeBlock id="RandomVector()[D:2"/>
      <jPACE:callMethod class="java.util.Random"
```



```
                        method="nextDouble" descriptor="()D"/>
      <jPACE:bytecodeBlock id="RandomVector()[D:3"/>

    </jPACE:loop>

    <jPACE:bytecodeBlock id="RandomVector()[D:4"/>

  </jPACE:method>

  <jPACE:method class="uk.ac.warwick.dcs.hpsg.applications.jgf.sparsematmult.JGF
SparseMatmultBench"
                method="JGFinitialise" descriptor="()V" type="characterised">

    <jPACE:bytecodeBlock id="JGFinitialise()V:1"/>

    <jPACE:MPIcase>
      <jPACE:probValue platform="${nP} - 1">

        <jPACE:bytecodeBlock id="JGFinitialise()V:2"/>

      </jPACE:probValue>
    </jPACE:MPIcase>

    <jPACE:bytecodeBlock id="JGFinitialise()V:3"/>
    <jPACE:callMethod class="uk.ac.warwick.dcs.hpsg.applications.jgf.sparsematmu
lt.JGFSparseMatmultBench"
                      method="RandomVector" descriptor="()[D"/>
    <jPACE:bytecodeBlock id="JGFinitialise()V:4"/>

    <jPACE:MPIcase>
      <jPACE:probValue platform="0">

        <jPACE:bytecodeBlock id="JGFinitialise()V:5"/>

      </jPACE:probValue>
    </jPACE:MPIcase>

    <jPACE:bytecodeBlock id="JGFinitialise()V:6"/>

    <jPACE:MPIcase>
      <jPACE:probValue platform="0">

        <jPACE:bytecodeBlock id="JGFinitialise()V:7"/>

        <jPACE:loop count="${p_datasizes_nz}">

          <jPACE:bytecodeBlock id="JGFinitialise()V:8"/>
          <jPACE:callMethod class="java.util.Random"
                            method="nextInt" descriptor="()I"/>
          <jPACE:bytecodeBlock id="JGFinitialise()V:9"/>
          <jPACE:callMethod class="java.lang.Math"
                            method="abs" descriptor="(I)I"/>
```

277

```
        <jPACE:bytecodeBlock id="JGFinitialise()V:10"/>
        <jPACE:callMethod class="java.util.Random"
                          method="nextInt" descriptor="()I"/>
        <jPACE:bytecodeBlock id="JGFinitialise()V:11"/>
        <jPACE:callMethod class="java.lang.Math"
                          method="abs" descriptor="(I)I"/>
        <jPACE:bytecodeBlock id="JGFinitialise()V:12"/>
        <jPACE:callMethod class="java.util.Random"
                          method="nextDouble" descriptor="()D"/>
        <jPACE:bytecodeBlock id="JGFinitialise()V:13"/>

      </jPACE:loop>

      <jPACE:bytecodeBlock id="JGFinitialise()V:14"/>

      <jPACE:loop count="${nP} - 1">

        <jPACE:bytecodeBlock id="JGFinitialise()V:15"/>

        <jPACE:case>
          <jPACE:probValue value="1 / (${nP} - 1)">

            <jPACE:bytecodeBlock id="JGFinitialise()V:16"/>

          </jPACE:probValue>
        </jPACE:case>

        <jPACE:bytecodeBlock id="JGFinitialise()V:17"/>

        <jPACE:loop count="${p_datasizes_nz}">

          <jPACE:bytecodeBlock id="JGFinitialise()V:18"/>
          <jPACE:callMethod class="java.util.Random"
                            method="nextInt" descriptor="()I"/>
          <jPACE:bytecodeBlock id="JGFinitialise()V:19"/>
          <jPACE:callMethod class="java.lang.Math"
                            method="abs" descriptor="(I)I"/>
          <jPACE:bytecodeBlock id="JGFinitialise()V:20"/>
          <jPACE:callMethod class="java.util.Random"
                            method="nextInt" descriptor="()I"/>
          <jPACE:bytecodeBlock id="JGFinitialise()V:21"/>
          <jPACE:callMethod class="java.lang.Math"
                            method="abs" descriptor="(I)I"/>
          <jPACE:bytecodeBlock id="JGFinitialise()V:22"/>
          <jPACE:callMethod class="java.util.Random"
                            method="nextDouble" descriptor="()D"/>
          <jPACE:bytecodeBlock id="JGFinitialise()V:23"/>

        </jPACE:loop>

        <jPACE:bytecodeBlock id="JGFinitialise()V:24"/>
        <jPACE:MPISsend destAPI="Recv" datatype="MPI.INT" size="${p_datasizes_
```



```
nz}"/>
          <jPACE:bytecodeBlock id="JGFinitialise()V:25"/>
          <jPACE:MPISsend destAPI="Recv" datatype="MPI.INT" size="${p_datasizes_
nz}"/>
          <jPACE:bytecodeBlock id="JGFinitialise()V:26"/>
          <jPACE:MPISsend destAPI="Recv" datatype="MPI.DOUBLE" size="${p_datasiz
es_nz}"/>
          <jPACE:bytecodeBlock id="JGFinitialise()V:27"/>

        </jPACE:loop>

        <jPACE:bytecodeBlock id="JGFinitialise()V:28"/>

      </jPACE:probValue>
      <jPACE:probValue platform="1 -- (${nP} - 1)">

        <jPACE:bytecodeBlock id="JGFinitialise()V:29"/>
        <jPACE:MPIRecv sourceAPI="Ssend" datatype="MPI.INT" size="${p_datasizes_
nz}"/>
        <jPACE:bytecodeBlock id="JGFinitialise()V:30"/>
        <jPACE:MPIRecv sourceAPI="Ssend" datatype="MPI.INT" size="${p_datasizes_
nz}"/>
        <jPACE:bytecodeBlock id="JGFinitialise()V:31"/>
        <jPACE:MPIRecv sourceAPI="Ssend" datatype="MPI.DOUBLE" size="${p_datasiz
es_nz}"/>
        <jPACE:bytecodeBlock id="JGFinitialise()V:32"/>

      </jPACE:probValue>
    </jPACE:MPIcase>

  </jPACE:method>

</jPACE:transaction>
```

```
<?xml version="1.0" encoding="UTF-8"?>

<jPACE:transaction>

  <jPACE:variable name="M"/>
  <jPACE:variable name="nz"/>
  <jPACE:variable name="SPARSE_NUM_ITER"/>
  <jPACE:variable name="p_datasizes_nz"/>

  <jPACE:confidence variable="c">
    <jPACE:setVariable variable="c" value="1"/>
  </jPACE:confidence>

  <jPACE:proc name="main">
    <jPACE:setVariable variable="p_datasizes_nz" value="(${nz} + ${nP} - 1) / ${
nP}"/>

    <jPACE:if leftExpression="${cP} - 1" condition="EQUALS" rightExpression="${n
P}">
      <jPACE:if leftExpression="${p_datasizes_nz}*${nP}" condition="GREATER_THAN
" rightExpression="${nz}">
        <jPACE:setVariable variable="p_datasizes_nz" value="${p_datasizes_nz} -
((${p_datasizes_nz}*${nP}) - ${nz})"/>
      </jPACE:if>
    </jPACE:if>

    <jPACE:evaluateMethod class="uk.ac.warwick.dcs.hpsg.applications.jgf.sparsem
atmult.JGFSparseMatmultBench"
                          method="JGFkernel" descriptor="()V"/>
  </jPACE:proc>

  <jPACE:method class="uk.ac.warwick.dcs.hpsg.applications.jgf.sparsematmult.Spa
rseMatmult"
                method="test" descriptor="([D[D[I[I[DI[I[D)V" type="characterise
d">

    <jPACE:bytecodeBlock id="test([D[D[I[I[DI[I[D)V:1"/>

    <jPACE:MPIBarrier/>

    <jPACE:bytecodeBlock id="test([D[D[I[I[DI[I[D)V:2"/>

    <jPACE:loop count="${SPARSE_NUM_ITER}">

      <jPACE:bytecodeBlock id="test([D[D[I[I[DI[I[D)V:3"/>

      <jPACE:loop count="${p_datasizes_nz}">

        <jPACE:bytecodeBlock id="test([D[D[I[I[DI[I[D)V:4"/>

      </jPACE:loop>
```



```
    </jPACE:loop>

    <jPACE:bytecodeBlock id="test([D[D[I[I[DI[I[D)V:5"/>

    <jPACE:MPIAllreduce datatype="MPI.DOUBLE" function="MPI.SUM" size="${M}"/>

    <jPACE:bytecodeBlock id="test([D[D[I[I[DI[I[D)V:6"/>
    <jPACE:bytecodeBlock id="test([D[D[I[I[DI[I[D)V:7"/>

    <jPACE:MPIBarrier/>

    <jPACE:bytecodeBlock id="test([D[D[I[I[DI[I[D)V:8"/>

    <jPACE:MPIcase>
      <jPACE:probValue platform="0">

        <jPACE:bytecodeBlock id="test([D[D[I[I[DI[I[D)V:9"/>

        <jPACE:loop count="${p_datasizes_nz}">

          <jPACE:bytecodeBlock id="test([D[D[I[I[DI[I[D)V:10"/>

        </jPACE:loop>

      </jPACE:probValue>
    </jPACE:MPIcase>

  </jPACE:method>

  <jPACE:method class="uk.ac.warwick.dcs.hpsg.applications.jgf.sparsematmult.JGF
SparseMatmultBench"
                method="JGFkernel" descriptor="()V" type="characterised">

    <jPACE:bytecodeBlock id="JGFkernel()V:1"/>
    <jPACE:callMethod class="uk.ac.warwick.dcs.hpsg.applications.jgf.sparsematmu
lt.SparseMatmult"
                      method="test" descriptor="([D[D[I[I[DI[I[D)V"/>

  </jPACE:method>

</jPACE:transaction>
```

279

```
<?xml version="1.0" encoding="UTF-8"?>

<jPACE:application>

  <jPACE:confidence max="1"/>

  <jPACE:platform resource="mscs-01.dcs.warwick.ac.uk" vm="sun-hotspot-i386-linu
x-1.4.1_01"/>
  <jPACE:platform resource="mscs-02.dcs.warwick.ac.uk" vm="sun-hotspot-i386-linu
x-1.4.1_01"/>
  <jPACE:platform resource="mscs-03.dcs.warwick.ac.uk" vm="sun-hotspot-i386-linu
x-1.4.1_01"/>
  <jPACE:platform resource="mscs-04.dcs.warwick.ac.uk" vm="sun-hotspot-i386-linu
x-1.4.1_01"/>
  <jPACE:platform resource="mscs-05.dcs.warwick.ac.uk" vm="sun-hotspot-i386-linu
x-1.4.1_01"/>
  <jPACE:platform resource="mscs-06.dcs.warwick.ac.uk" vm="sun-hotspot-i386-linu
x-1.4.1_01"/>
  <jPACE:platform resource="mscs-07.dcs.warwick.ac.uk" vm="sun-hotspot-i386-linu
x-1.4.1_01"/>
  <jPACE:platform resource="mscs-08.dcs.warwick.ac.uk" vm="sun-hotspot-i386-linu
x-1.4.1_01"/>
  <jPACE:platform resource="mscs-09.dcs.warwick.ac.uk" vm="sun-hotspot-i386-linu
x-1.4.1_01"/>
  <jPACE:platform resource="mscs-10.dcs.warwick.ac.uk" vm="sun-hotspot-i386-linu
x-1.4.1_01"/>
  <jPACE:platform resource="mscs-11.dcs.warwick.ac.uk" vm="sun-hotspot-i386-linu
x-1.4.1_01"/>
  <jPACE:platform resource="mscs-12.dcs.warwick.ac.uk" vm="sun-hotspot-i386-linu
x-1.4.1_01"/>
  <jPACE:platform resource="mscs-13.dcs.warwick.ac.uk" vm="sun-hotspot-i386-linu
x-1.4.1_01"/>
  <jPACE:platform resource="mscs-14.dcs.warwick.ac.uk" vm="sun-hotspot-i386-linu
x-1.4.1_01"/>
  <jPACE:platform resource="mscs-15.dcs.warwick.ac.uk" vm="sun-hotspot-i386-linu
x-1.4.1_01"/>
  <jPACE:platform resource="mscs-16.dcs.warwick.ac.uk" vm="sun-hotspot-i386-linu
x-1.4.1_01"/>

  <jPACE:parameter name="array_rows" value="100000"/>

  <jPACE:link targetObject="series.tranmap" targetVariable="array_rows" newValue
="${array_rows}"/>

  <jPACE:proc name="main">
    <jPACE:evaluateTransactionMap transactionMap="series.tranmap" platforms="ALL
"/>
  </jPACE:proc>

</jPACE:application>
```

```
<?xml version="1.0" encoding="UTF-8"?>

<jPACE:transactionMap>

  <jPACE:variable name="array_rows"/>

  <jPACE:link targetObject="series-kernel.tran" targetVariable="array_rows" newV
alue="${array_rows}"/>

  <jPACE:proc name="main">
    <jPACE:evaluateMap map="series.map"/>
  </jPACE:proc>

  <jPACE:map name="series.map">

    <jPACE:step>
      <jPACE:evaluateTransaction transaction="series-kernel.tran" platforms="ALL
"/>
    </jPACE:step>

  </jPACE:map>

</jPACE:transactionMap>
```

```
<?xml version="1.0" encoding="UTF-8"?>

<jPACE:transaction>

  <jPACE:variable name="array_rows"/>
  <jPACE:variable name="p_array_rows"/>

  <jPACE:confidence variable="c">
    <jPACE:setVariable variable="c" value="1"/>
  </jPACE:confidence>

  <jPACE:proc name="main">
    <jPACE:setVariable variable="p_array_rows" value="(${array_rows} + ${nP} - 1
) / ${nP}"/>

    <jPACE:if leftExpression="${cP}" condition="EQUALS" rightExpression="${nP} -
 1">
      <jPACE:if leftExpression="${p_array_rows}*${nP}" condition="GREATER_THAN"
rightExpression="${array_rows}">
        <jPACE:setVariable variable="p_array_rows" value="${p_array_rows} - ((${
p_array_rows}*${nP}) - ${array_rows})"/>
      </jPACE:if>
    </jPACE:if>

    <jPACE:evaluateMethod class="uk.ac.warwick.dcs.hpsg.applications.jgf.series.
JGFSeriesBench"
                          method="JGFkernel" descriptor="()V"/>
  </jPACE:proc>

  <jPACE:method class="java.lang.Math"
                method="sin" descriptor="(D)D" type="transaction">
  </jPACE:method>

  <jPACE:method class="java.lang.Math"
                method="cos" descriptor="(D)D" type="transaction">
  </jPACE:method>

  <jPACE:method class="java.lang.Math"
                method="pow" descriptor="(DD)D" type="transaction">
  </jPACE:method>

  <jPACE:method class="uk.ac.warwick.dcs.hpsg.applications.jgf.series.SeriesTest
"
                method="thefunction" descriptor="(DDI)D" type="characterised">

    <jPACE:bytecodeBlock id="thefunction(DDI)D:1"/>

    <jPACE:case>
      <jPACE:probValue value="1 / ((2 * ${p_array_rows}) + 1)">

        <jPACE:bytecodeBlock id="thefunction(DDI)D:2"/>
        <jPACE:callMethod class="java.lang.Math"
```



```
                          method="pow" descriptor="(DD)D"/>
        <jPACE:bytecodeBlock id="thefunction(DDI)D:3"/>

      </jPACE:probValue>
      <jPACE:probValue value="${p_array_rows} / ((2 * ${p_array_rows}) + 1)">

        <jPACE:bytecodeBlock id="thefunction(DDI)D:4"/>
        <jPACE:callMethod class="java.lang.Math"
                          method="pow" descriptor="(DD)D"/>
        <jPACE:bytecodeBlock id="thefunction(DDI)D:5"/>
        <jPACE:callMethod class="java.lang.Math"
                          method="cos" descriptor="(D)D"/>
        <jPACE:bytecodeBlock id="thefunction(DDI)D:6"/>

      </jPACE:probValue>
      <jPACE:probValue value="${p_array_rows} / ((2 * ${p_array_rows}) + 1)">

        <jPACE:bytecodeBlock id="thefunction(DDI)D:7"/>
        <jPACE:callMethod class="java.lang.Math"
                          method="pow" descriptor="(DD)D"/>
        <jPACE:bytecodeBlock id="thefunction(DDI)D:8"/>
        <jPACE:callMethod class="java.lang.Math"
                          method="sin" descriptor="(D)D"/>
        <jPACE:bytecodeBlock id="thefunction(DDI)D:9"/>

      </jPACE:probValue>
      <jPACE:probValue value="0">

        <jPACE:bytecodeBlock id="thefunction(DDI)D:10"/>

      </jPACE:probValue>
    </jPACE:case>

    <jPACE:bytecodeBlock id="thefunction(DDI)D:11"/>

  </jPACE:method>

  <jPACE:method class="uk.ac.warwick.dcs.hpsg.applications.jgf.series.SeriesTest
"
                method="TrapezoidIntegrate" descriptor="(DDIDI)D" type="characte
rised">

    <jPACE:bytecodeBlock id="TrapezoidIntegrate(DDIDI)D:1"/>
    <jPACE:callMethod class="uk.ac.warwick.dcs.hpsg.applications.jgf.series.Seri
esTest"
                      method="thefunction" descriptor="(DDI)D"/>
    <jPACE:bytecodeBlock id="TrapezoidIntegrate(DDIDI)D:2"/>

    <jPACE:case>
      <jPACE:probValue value="1">

        <jPACE:bytecodeBlock id="TrapezoidIntegrate(DDIDI)D:3"/>
```

```
        <jPACE:loop count="999">

          <jPACE:bytecodeBlock id="TrapezoidIntegrate(DDIDI)D:4"/>
          <jPACE:callMethod class="uk.ac.warwick.dcs.hpsg.applications.jgf.serie
s.SeriesTest"
                            method="thefunction" descriptor="(DDI)D"/>
          <jPACE:bytecodeBlock id="TrapezoidIntegrate(DDIDI)D:5"/>

        </jPACE:loop>

      </jPACE:probValue>
    </jPACE:case>

    <jPACE:bytecodeBlock id="TrapezoidIntegrate(DDIDI)D:6"/>
    <jPACE:callMethod class="uk.ac.warwick.dcs.hpsg.applications.jgf.series.Seri
esTest"
                      method="thefunction" descriptor="(DDI)D"/>
    <jPACE:bytecodeBlock id="TrapezoidIntegrate(DDIDI)D:7"/>

  </jPACE:method>

  <jPACE:method class="uk.ac.warwick.dcs.hpsg.applications.jgf.series.JGFSeriesB
ench"
                method="Do" descriptor="()V" type="characterised">

    <jPACE:bytecodeBlock id="Do()V:1"/>

    <jPACE:MPIcase>
      <jPACE:probValue platform="0">

        <jPACE:bytecodeBlock id="Do()V:2"/>
        <jPACE:callMethod class="uk.ac.warwick.dcs.hpsg.applications.jgf.series.
SeriesTest"
                          method="TrapezoidIntegrate" descriptor="(DDIDI)D"/>
        <jPACE:bytecodeBlock id="Do()V:3"/>

      </jPACE:probValue>
    </jPACE:MPIcase>

    <jPACE:bytecodeBlock id="Do()V:4"/>

    <jPACE:MPIcase>
      <jPACE:probValue platform="0">

        <jPACE:bytecodeBlock id="Do()V:5"/>

      </jPACE:probValue>
      <jPACE:probValue platform="1 -- (${nP} - 1)">

        <jPACE:bytecodeBlock id="Do()V:6"/>
```



```
      </jPACE:probValue>
    </jPACE:MPIcase>

    <jPACE:bytecodeBlock id="Do()V:7"/>

    <jPACE:MPIcase>
      <jPACE:probValue platform="0">
        <jPACE:loop count="${p_array_rows} - 1">

          <jPACE:bytecodeBlock id="Do()V:8"/>
          <jPACE:callMethod class="uk.ac.warwick.dcs.hpsg.applications.jgf.serie
s.SeriesTest"
                            method="TrapezoidIntegrate" descriptor="(DDIDI)D"/>
          <jPACE:bytecodeBlock id="Do()V:9"/>
          <jPACE:callMethod class="uk.ac.warwick.dcs.hpsg.applications.jgf.serie
s.SeriesTest"
                            method="TrapezoidIntegrate" descriptor="(DDIDI)D"/>
          <jPACE:bytecodeBlock id="Do()V:10"/>

        </jPACE:loop>
      </jPACE:probValue>
      <jPACE:probValue platform="1 -- (${nP} - 1)">
        <jPACE:loop count="${p_array_rows}">

          <jPACE:bytecodeBlock id="Do()V:8"/>
          <jPACE:callMethod class="uk.ac.warwick.dcs.hpsg.applications.jgf.serie
s.SeriesTest"
                            method="TrapezoidIntegrate" descriptor="(DDIDI)D"/>
          <jPACE:bytecodeBlock id="Do()V:9"/>
          <jPACE:callMethod class="uk.ac.warwick.dcs.hpsg.applications.jgf.serie
s.SeriesTest"
                            method="TrapezoidIntegrate" descriptor="(DDIDI)D"/>
          <jPACE:bytecodeBlock id="Do()V:10"/>

        </jPACE:loop>
      </jPACE:probValue>
    </jPACE:MPIcase>

    <jPACE:bytecodeBlock id="Do()V:11"/>

    <jPACE:MPIBarrier/>

    <jPACE:bytecodeBlock id="Do()V:12"/>

    <jPACE:MPIcase>
      <jPACE:probValue platform="0'>

        <jPACE:bytecodeBlock id="Do()V:13"/>

        <jPACE:loop count="${p_array_rows} - 1'>

          <jPACE:bytecodeBlock id="Do()V:14"/>
```

```
        </jPACE:loop>

        <jPACE:bytecodeBlock id="Do()V:15"/>

        <jPACE:loop count="${nP} - 1">

          <jPACE:bytecodeBlock id="Do()V:16"/>
          <jPACE:MPIRecv sourceAPI="Ssend" datatype="MPI.DOUBLE" size="${p_array
_rows}"/>
          <jPACE:bytecodeBlock id="Do()V:17"/>
          <jPACE:MPIRecv sourceAPI="Ssend" datatype="MPI.DOUBLE" size="${p_array
_rows}"/>
          <jPACE:bytecodeBlock id="Do()V:18"/>

          <jPACE:case>
            <jPACE:probValue value="1 / (${nP} - 1)">

              <jPACE:bytecodeBlock id="Do()V:19"/>

            </jPACE:probValue>
          </jPACE:case>

          <jPACE:bytecodeBlock id="Do()V:20"/>

          <jPACE:loop count="${p_array_rows}">

            <jPACE:bytecodeBlock id="Do()V:21"/>

          </jPACE:loop>

          <jPACE:bytecodeBlock id="Do()V:22"/>

        </jPACE:loop>

        <jPACE:bytecodeBlock id="Do()V:23"/>

      </jPACE:probValue>
      <jPACE:probValue platform="1 -- (${nP} - 1)">

        <jPACE:bytecodeBlock id="Do()V:24"/>
        <jPACE:MPISsend destAPI="Recv" datatype="MPI.DOUBLE" size="${p_array_row
s}"/>
        <jPACE:bytecodeBlock id="Do()V:25"/>
        <jPACE:MPISsend destAPI="Recv" datatype="MPI.DOUBLE" size="${p_array_row
s}"/>

      </jPACE:probValue>
    </jPACE:MPIcase>

    <jPACE:bytecodeBlock id="Do()V:26"/>
    <jPACE:MPIBarrier/>
```



```
  </jPACE:method>

  <jPACE:method class="uk.ac.warwick.dcs.hpsg.applications.jgf.series.JGFSeriesB
ench"
                method="JGFkernel" descriptor="()V" type="characterised">

    <jPACE:bytecodeBlock id="JGFkernel()V:1"/>
    <jPACE:callMethod class="uk.ac.warwick.dcs.hpsg.applications.jgf.series.JGFS
eriesBench"
                      method="Do" descriptor="()V"/>

  </jPACE:method>

</jPACE:transaction>
```

```
<?xml version="1.0" encoding="UTF-8"?>

<jPACE:application>

  <jPACE:confidence max="1"/>

  <jPACE:platform resource="mscs-01.dcs.warwick.ac.uk" vm="sun-hotspot-i386-linu
x-1.4.1_01"/>
  <jPACE:platform resource="mscs-02.dcs.warwick.ac.uk" vm="sun-hotspot-i386-linu
x-1.4.1_01"/>
  <jPACE:platform resource="mscs-03.dcs.warwick.ac.uk" vm="sun-hotspot-i386-linu
x-1.4.1_01"/>
  <jPACE:platform resource="mscs-04.dcs.warwick.ac.uk" vm="sun-hotspot-i386-linu
x-1.4.1_01"/>
  <jPACE:platform resource="mscs-05.dcs.warwick.ac.uk" vm="sun-hotspot-i386-linu
x-1.4.1_01"/>
  <jPACE:platform resource="mscs-06.dcs.warwick.ac.uk" vm="sun-hotspot-i386-linu
x-1.4.1_01"/>
  <jPACE:platform resource="mscs-07.dcs.warwick.ac.uk" vm="sun-hotspot-i386-linu
x-1.4.1_01"/>
  <jPACE:platform resource="mscs-08.dcs.warwick.ac.uk" vm="sun-hotspot-i386-linu
x-1.4.1_01"/>
  <jPACE:platform resource="mscs-09.dcs.warwick.ac.uk" vm="sun-hotspot-i386-linu
x-1.4.1_01"/>
  <jPACE:platform resource="mscs-10.dcs.warwick.ac.uk" vm="sun-hotspot-i386-linu
x-1.4.1_01"/>
  <jPACE:platform resource="mscs-11.dcs.warwick.ac.uk" vm="sun-hotspot-i386-linu
x-1.4.1_01"/>
  <jPACE:platform resource="mscs-12.dcs.warwick.ac.uk" vm="sun-hotspot-i386-linu
x-1.4.1_01"/>
  <jPACE:platform resource="mscs-13.dcs.warwick.ac.uk" vm="sun-hotspot-i386-linu
x-1.4.1_01"/>
  <jPACE:platform resource="mscs-14.dcs.warwick.ac.uk" vm="sun-hotspot-i386-linu
x-1.4.1_01"/>
  <jPACE:platform resource="mscs-15.dcs.warwick.ac.uk" vm="sun-hotspot-i386-linu
x-1.4.1_01"/>
  <jPACE:platform resource="mscs-16.dcs.warwick.ac.uk" vm="sun-hotspot-i386-linu
x-1.4.1_01"/>

  <jPACE:parameter name="array_rows" value="1000000"/>

  <jPACE:link targetObject="crypt.tranmap" targetVariable="array_rows" newValue=
"${array_rows}"/>

  <jPACE:proc name="main">
    <jPACE:evaluateTransactionMap transactionMap="crypt.tranmap" platforms="ALL"
/>
  </jPACE:proc>

</jPACE:application>
```

```
<?xml version="1.0" encoding="UTF-8"?>

<jPACE:transactionMap>

  <jPACE:variable name="array_rows"/>
  <jPACE:variable name="p_array_rows"/>

  <jPACE:variable name="p_array_rows_general"/>
  <jPACE:variable name="p_array_rows_last"/>

  <jPACE:link targetObject="crypt-init.tran" targetVariable="p_array_rows" newVa
lue="${p_array_rows}"/>
  <jPACE:link targetObject="crypt-init.tran" targetVariable="nprocess" newValue=
"${nP}"/>
  <jPACE:link targetObject="crypt-encrypt.tran" targetVariable="p_array_rows" ne
wValue="${p_array_rows}"/>
  <jPACE:link targetObject="crypt-finalise.tran" targetVariable="p_array_rows" n
ewValue="${p_array_rows}"/>

  <jPACE:proc name="main">
    <jPACE:setVariable variable="p_array_rows_general" value="(((${array_rows} /
 8) + ${nP} - 1) / ${nP}) * 8"/>
    <jPACE:setVariable variable="p_array_rows_last" value="${p_array_rows_genera
l}"/>

    <jPACE:if leftExpression="${p_array_rows_general}*${nP}" condition="GREATER_
THAN" rightExpression="${array_rows}">
      <jPACE:setVariable variable="p_array_rows_last" value="${p_array_rows_gene
ral} - ((${p_array_rows_general}*${nP}) - ${array_rows})"/>
    </jPACE:if>

    <jPACE:evaluateMap map="crypt.map"/>
  </jPACE:proc>

  <jPACE:map name="crypt.map">

    <jPACE:setVariable variable="p_array_rows" value="${p_array_rows_general}"/>
    <jPACE:step>
      <jPACE:evaluateTransaction transaction="crypt-init.tran" platforms="0"/>
    </jPACE:step>

    <jPACE:for variable="i" startValue="1" endValue="${nP} - 2" increment="${i}
+ 1">
      <jPACE:step>
        <jPACE:MPISsend destAPI="Recv" source="0" dest="${i}" datatype="MPI.BYTE
" size="${p_array_rows_general}"/>
      </jPACE:step>
    </jPACE:for>
    <jPACE:step>
      <jPACE:MPISsend destAPI="Recv" source="0" dest="${nP} - 1" datatype="MPI.B
YTE" size="${p_array_rows_last}"/>
    </jPACE:step>
```



```
    <jPACE:for variable="i" startValue="1" endValue="2" increment="${i} + 1">
      <jPACE:step>
        <jPACE:setVariable variable="p_array_rows" value="${p_array_rows_general
}"/>
        <jPACE:evaluateTransaction transaction="crypt-encrypt.tran" platforms="0
 -- (${nP} - 2)"/>

        <jPACE:setVariable variable="p_array_rows" value="${p_array_rows_last}"/
>
        <jPACE:evaluateTransaction transaction="crypt-encrypt.tran" platforms="$
{nP} - 1"/>
      </jPACE:step>
    </jPACE:for>

    <jPACE:setVariable variable="p_array_rows" value="${p_array_rows_general}"/>
    <jPACE:step>
      <jPACE:evaluateTransaction transaction="crypt-finalise.tran" platforms="0"
/>
    </jPACE:step>

    <jPACE:for variable="i" startValue="1" endValue="${nP} - 2" increment="${i}
+ 1">
      <jPACE:step>
        <jPACE:MPIRecv sourceAPI="Ssend" source="${i}" dest="0" datatype="MPI.BY
TE" size="${p_array_rows_general}"/>
      </jPACE:step>
    </jPACE:for>

    <jPACE:step>
      <jPACE:MPIRecv sourceAPI="Ssend" source="${nP} - 1" dest="0" datatype="MPI
.BYTE" size="${p_array_rows_last}"/>
    </jPACE:step>

  </jPACE:map>

</jPACE:transactionMap>
```

```
<?xml version="1.0" encoding="UTF-8"?>

<jPACE:transaction>

  <jPACE:variable name="p_array_rows"/>
  <jPACE:variable name="nprocess"/>

  <jPACE:confidence variable="c">
    <jPACE:setVariable variable="c" value="1"/>
  </jPACE:confidence>

  <jPACE:proc name="main">
    <jPACE:evaluateMethod class="uk.ac.warwick.dcs.hpsg.applications.jgf.crypt.J
GFCryptBench"
                          method="Do" descriptor="()V"/>
  </jPACE:proc>

  <jPACE:method class="uk.ac.warwick.dcs.hpsg.applications.jgf.crypt.JGFCryptBen
ch"
                method="Do" descriptor="()V" type="characterised">

    <jPACE:bytecodeBlock id="Do()V:1"/>
    <jPACE:bytecodeBlock id="Do()V:2"/>
    <jPACE:bytecodeBlock id="Do()V:3"/>

    <jPACE:loop count="${p_array_rows}">

      <jPACE:bytecodeBlock id="Do()V:4"/>

    </jPACE:loop>

    <jPACE:bytecodeBlock id="Do()V:5"/>

    <jPACE:loop count="${nprocess} - 1">

      <jPACE:bytecodeBlock id="Do()V:6"/>

      <jPACE:case>
        <jPACE:probValue value="1 / (${nprocess} - 1)">

          <jPACE:bytecodeBlock id="Do()V:7"/>

        </jPACE:probValue>
        <jPACE:probValue value="1 - (1 / (${nprocess} - 1))">

          <jPACE:bytecodeBlock id="Do()V:8"/>

        </jPACE:probValue>
      </jPACE:case>

    </jPACE:loop>
```



```
    <jPACE:bytecodeBlock id="Do()V:11"/>

  </jPACE:method>

</jPACE:transaction>
```

```
<?xml version="1.0" encoding="UTF-8"?>

<jPACE:transaction>

  <jPACE:variable name="p_array_rows"/>

  <jPACE:confidence variable="c">
    <jPACE:setVariable variable="c" value="1"/>
  </jPACE:confidence>

  <jPACE:proc name="main">
    <jPACE:evaluateMethod class="uk.ac.warwick.dcs.hpsg.applications.jgf.crypt.I
DEATest"
                          method="cipher_idea" descriptor="([B[B[I)V"/>
  </jPACE:proc>

  <jPACE:method class="uk.ac.warwick.dcs.hpsg.applications.jgf.crypt.IDEATest"
                method="cipher_idea" descriptor="([B[B[I)V" type="characterised"
>

    <jPACE:bytecodeBlock id="cipher_idea([B[B[I)V:1"/>

    <jPACE:loop count="${p_array_rows} / 8">

      <jPACE:bytecodeBlock id="cipher_idea([B[B[I)V:2"/>

      <!-- this loop characterises a do_whle statement and therefore
           the previous block includes one of these iterations -->
      <jPACE:loop count="7">

        <jPACE:bytecodeBlock id="cipher_idea([B[B[I)V:3"/>

      </jPACE:loop>

      <jPACE:bytecodeBlock id="cipher_idea([B[B[I)V:4"/>

    </jPACE:loop>

  </jPACE:method>

</jPACE:transaction>
```

```
<?xml version="1.0" encoding="UTF-8"?>

<jPACE:transaction>

  <jPACE:variable name="p_array_rows"/>

  <jPACE:confidence variable="c">
    <jPACE:setVariable variable="c" value="1"/>
  </jPACE:confidence>

  <jPACE:proc name="main">
    <jPACE:evaluateMethod class="uk.ac.warwick.dcs.hpsg.applications.jgf.crypt.J
GFCryptBench"
                          method="Do" descriptor="()V"/>
  </jPACE:proc>

  <jPACE:method class="uk.ac.warwick.dcs.hpsg.applications.jgf.crypt.JGFCryptBen
ch"
                               method="Do" descriptor="()V" type="characterised
">

    <jPACE:bytecodeBlock id="Do()V:18"/>

    <jPACE:loop count="${p_array_rows}">

      <jPACE:bytecodeBlock id="Do()V:19"/>

    </jPACE:loop>

  </jPACE:method>

</jPACE:transaction>
```

```
<?xml version="1.0" encoding="UTF-8"?>

<jPACE:application>

  <jPACE:confidence max="1"/>

  <jPACE:platform resource="labvista.dcs.warwick.ac.uk" vm="sun-hotspot-i386-lin
ux-1.4.1_01"/>

  <jPACE:parameter name="n" value="24"/>

  <jPACE:link targetObject="ep.tranmap" targetVariable="n" newValue="2^${n}"/>

  <jPACE:proc name="main">
    <jPACE:evaluateTransactionMap transactionMap="ep.tranmap" platforms="ALL"/>
  </jPACE:proc>

</jPACE:application>
```

```
<?xml version="1.0" encoding="UTF-8"?>

<jPACE:transactionMap>

  <jPACE:variable name="n"/>

  <jPACE:link targetObject="ep-kernel.tran" targetVariable="n" newValue="${n}"/>

  <jPACE:proc name="main">
    <jPACE:evaluateMap map="kernel"/>
  </jPACE:proc>

  <jPACE:map name="kernel">

    <jPACE:step>
      <jPACE:evaluateTransaction transaction="ep-kernel.tran" platforms="ALL"/>
    </jPACE:step>

  </jPACE:map>

</jPACE:transactionMap>
```

```
<?xml version="1.0" encoding="UTF-8"?>

<jPACE:transaction>

  <jPACE:variable name="n"/>

  <jPACE:confidence variable="c">
    <jPACE:setVariable variable="c" value="${nE} / 5"/>
  </jPACE:confidence>

  <jPACE:proc name="main">
    <jPACE:evaluateMethod class="uk.ac.warwick.dcs.hpsg.applications.dhpc.ep.DHP
C_EPBench"
                          method="JGFkernel" descriptor="()V"/>
  </jPACE:proc>

  <jPACE:method class="uk.ac.warwick.dcs.hpsg.applications.dhpc.ep.DHPC_EPBench"
                method="ep" descriptor="()V" type="characterised">

    <jPACE:bytecodeBlock id="ep()V:1"/>

    <jPACE:loop count="10">

      <jPACE:bytecodeBlock id="ep()V:2"/>

    </jPACE:loop>

    <jPACE:bytecodeBlock id="ep()V:3"/>

    <jPACE:loop count="${n}">

      <jPACE:bytecodeBlock id="ep()V:4"/>
      <jPACE:callMethod class="uk.ac.warwick.dcs.hpsg.applications.dhpc.ep.Kerne
lEP"
                        method="next" descriptor="()D"/>
      <jPACE:bytecodeBlock id="ep()V:5"/>
      <jPACE:callMethod class="uk.ac.warwick.dcs.hpsg.applications.dhpc.ep.Kerne
lEP"
                        method="next" descriptor="()D"/>
      <jPACE:bytecodeBlock id="ep()V:6"/>

      <jPACE:case>
        <jPACE:probValue value="0.78372" data_dependent="yes">

          <jPACE:bytecodeBlock id="ep()V:7"/>
          <jPACE:callMethod class="java.lang.Math"
                            method="log" descriptor="(D)D"/>
          <jPACE:bytecodeBlock id="ep()V:8"/>
          <jPACE:callMethod class="java.lang.Math"
                            method="sqrt" descriptor="(D)D"/>
          <jPACE:bytecodeBlock id="ep()V:9"/>
          <jPACE:callMethod class="java.lang.Math"
```



```
                            method="abs" descriptor="(D)D"/>
          <jPACE:bytecodeBlock id="ep()V:10"/>
          <jPACE:callMethod class="java.lang.Math"
                            method="abs" descriptor="(D)D"/>
          <jPACE:bytecodeBlock id="ep()V:11"/>
          <jPACE:callMethod class="java.lang.Math"
                            method="max" descriptor="(DD)D"/>
          <jPACE:bytecodeBlock id="ep()V:12"/>
          <jPACE:callMethod class="java.lang.Math"
                            method="floor" descriptor="(D)D"/>
          <jPACE:bytecodeBlock id="ep()V:13"/>

        </jPACE:probValue>
      </jPACE:case>

      <jPACE:bytecodeBlock id="ep()V:14"/>

    </jPACE:loop>

  </jPACE:method>

  <jPACE:method class="java.lang.Math"
                method="floor" descriptor="(D)D" type="transaction">
  </jPACE:method>

  <jPACE:method class="java.lang.Math"
                method="max" descriptor="(DD)D" type="transaction">
  </jPACE:method>

  <jPACE:method class="java.lang.Math"
                method="abs" descriptor="(D)D" type="transaction">
  </jPACE:method>

  <jPACE:method class="java.lang.Math"
                method="sqrt" descriptor="(D)D" type="transaction">
  </jPACE:method>

  <jPACE:method class="java.lang.Math"
                method="log" descriptor="(D)D" type="transaction">
  </jPACE:method>

  <jPACE:method class="uk.ac.warwick.dcs.hpsg.applications.dhpc.ep.KernelEP"
                method="next" descriptor="()D" type="characterised">

    <jPACE:bytecodeBlock id="next()D:1"/>

    <jPACE:case>
      <jPACE:probValue value="1 / ${n}">

        <jPACE:bytecodeBlock id="next()D:2"/>

        <jPACE:loop count="23">
```

292

```
        <jPACE:bytecodeBlock id="next()D:3"/>

      </jPACE:loop>

      <jPACE:bytecodeBlock id="next()D:4"/>

      <jPACE:loop count="46">

        <jPACE:bytecodeBlock id="next()D:5"/>

      </jPACE:loop>

      <jPACE:bytecodeBlock id="next()D:6"/>

    </jPACE:probValue>
  </jPACE:case>

  <jPACE:bytecodeBlock id="next()D:7"/>

</jPACE:method>

<jPACE:method class="uk.ac.warwick.dcs.hpsg.applications.dhpc.ep.DHPC_EPBench"
              method="JGFkernel" descriptor="()V" type="characterised">

  <jPACE:bytecodeBlock id="JGFkernel()V:1"/>
  <jPACE:callMethod class="uk.ac.warwick.dcs.hpsg.applications.dhpc.ep.DHPC_EP
Bench"
                    method="ep" descriptor="()V"/>

</jPACE:method>

</jPACE:transaction>
```

```
<?xml version="1.0" encoding="UTF-8"?>

<jPACE:application>

  <jPACE:confidence max="1"/>

  <jPACE:platform resource="budweiser.dcs.warwick.ac.uk" vm="sun-hotspot-sparc-s
olaris-1.4.1_01"/>

  <jPACE:parameter name="n1" value="64"/>
  <jPACE:parameter name="n2" value="64"/>
  <jPACE:parameter name="n3" value="64"/>

  <jPACE:link targetObject="fft.tranmap" targetVariable="n1" newValue="${n1}"/>
  <jPACE:link targetObject="fft.tranmap" targetVariable="n2" newValue="${n2}"/>
  <jPACE:link targetObject="fft.tranmap" targetVariable="n3" newValue="${n3}"/>

  <jPACE:proc name="main">
    <jPACE:evaluateTransactionMap transactionMap="fft.tranmap" platforms="ALL"/>
  </jPACE:proc>

</jPACE:application>
```

```
<?xml version="1.0" encoding="UTF-8"?>

<jPACE:transactionMap>

  <jPACE:variable name="n1"/>
  <jPACE:variable name="n2"/>
  <jPACE:variable name="n3"/>

  <jPACE:link targetObject="fft-kernel.tran" targetVariable="n1" newValue="${n1}
"/>
  <jPACE:link targetObject="fft-kernel.tran" targetVariable="n2" newValue="${n2}
"/>
  <jPACE:link targetObject="fft-kernel.tran" targetVariable="n3" newValue="${n3}
"/>

  <jPACE:proc name="main">
    <jPACE:evaluateMap map="kernel"/>
  </jPACE:proc>

  <jPACE:map name="kernel">

    <jPACE:step>
      <jPACE:evaluateTransaction transaction="fft-kernel.tran" platforms="ALL"/>
    </jPACE:step>

  </jPACE:map>

</jPACE:transactionMap>
```

```
<?xml version="1.0" encoding="UTF-8"?>

<jPACE:transaction>

  <jPACE:variable name="n1"/>
  <jPACE:variable name="n2"/>
  <jPACE:variable name="n3"/>

  <jPACE:confidence variable="c">
    <jPACE:setVariable variable="c" value="${nE} / 5"/>
  </jPACE:confidence>

  <jPACE:proc name="main">

    <jPACE:evaluateMethod class="uk.ac.warwick.dcs.hpsg.applications.dhpc.fft.DH
PC_FFTBench"
                          method="JGFkernel" descriptor="()V"/>

  </jPACE:proc>

  <jPACE:method class="uk.ac.warwick.dcs.hpsg.applications.dhpc.fft.DHPC_FFTBenc
h"
                method="doFFT" descriptor="()V" type="characterised">

    <jPACE:bytecodeBlock id="doFFT()V:1"/>

    <jPACE:loop count="${n1} * ${n2} * ${n3} * 2">

      <jPACE:bytecodeBlock id="doFFT()V:2"/>
      <jPACE:callMethod class="uk.ac.warwick.dcs.hpsg.applications.dhpc.fft.DHPC
_FFTBench"
                        method="next" descriptor="()D"/>
      <jPACE:bytecodeBlock id="doFFT()V:3"/>

    </jPACE:loop>

    <jPACE:bytecodeBlock id="doFFT()V:4"/>
    <jPACE:callMethod class="uk.ac.warwick.dcs.hpsg.applications.dhpc.fft.DHPC_F
FTBench"
                      method="complex_fouriernd" descriptor="([D[I)V"/>
    <jPACE:bytecodeBlock id="doFFT()V:5"/>

    <jPACE:loop count="6">

      <jPACE:bytecodeBlock id="doFFT()V:6"/>
      <jPACE:callMethod class="uk.ac.warwick.dcs.hpsg.applications.dhpc.fft.DHPC
_FFTBench"
                        method="evolve" descriptor="([D[II)V"/>
      <jPACE:bytecodeBlock id="doFFT()V:7"/>
      <jPACE:callMethod class="uk.ac.warwick.dcs.hpsg.applications.dhpc.fft.DHPC
_FFTBench"
                        method="inverse_complex_fouriernd" descriptor="([D[I)V"/
```



```
>
      <jPACE:bytecodeBlock id="doFFT()V:8"/>

      <jPACE:loop count="1024">

        <jPACE:bytecodeBlock id="doFFT()V:9"/>

      </jPACE:loop>

      <jPACE:bytecodeBlock id="doFFT()V:10"/>

    </jPACE:loop>


  </jPACE:method>

  <jPACE:method class="uk.ac.warwick.dcs.hpsg.applications.dhpc.fft.DHPC_FFTBenc
h"
                method="inverse_complex_fouriernd" descriptor="([D[I)V" type="ch
aracterised">

    <jPACE:bytecodeBlock id="inverse_complex_fouriernd([D[I)V:1"/>
    <jPACE:callMethod class="uk.ac.warwick.dcs.hpsg.applications.dhpc.fft.DHPC_F
FTBench"
                      method="auxiliary_complex_fouriernd" descriptor="([D[II)V"
/>
    <jPACE:bytecodeBlock id="inverse_complex_fouriernd([D[I)V:2"/>

    <jPACE:loop count="${n1} * ${n2} * ${n3} * 2">

      <jPACE:bytecodeBlock id="inverse_complex_fouriernd([D[I)V:3"/>

    </jPACE:loop>


  </jPACE:method>

  <jPACE:method class="uk.ac.warwick.dcs.hpsg.applications.dhpc.fft.DHPC_FFTBenc
h"
                method="evolve" descriptor="([D[II)V" type="characterised">

    <jPACE:bytecodeBlock id="evolve([D[II)V:1"/>

    <jPACE:loop count="${n1}">

      <jPACE:bytecodeBlock id="evolve([D[II)V:2"/>

      <jPACE:loop count="${n2}">

        <jPACE:bytecodeBlock id="evolve([D[II)V:3"/>

        <jPACE:loop count="${n3}">
```

```
          <jPACE:bytecodeBlock id="evolve([D[II)V:4"/>
          <jPACE:callMethod class="uk.ac.warwick.dcs.hpsg.applications.dhpc.fft.
DHPC_FFTBench"
                            method="map" descriptor="(III)D"/>
          <jPACE:bytecodeBlock id="evolve([D[II)V:5"/>
          <jPACE:callMethod class="java.lang.Math"
                            method="exp" descriptor="(D)D"/>
          <jPACE:bytecodeBlock id="evolve([D[II)V:6"/>
          <jPACE:callMethod class="uk.ac.warwick.dcs.hpsg.applications.dhpc.fft.
DHPC_FFTBench"
                            method="map" descriptor="(III)D"/>
          <jPACE:bytecodeBlock id="evolve([D[II)V:7"/>
          <jPACE:callMethod class="java.lang.Math"
                            method="exp" descriptor="(D)D"/>
          <jPACE:bytecodeBlock id="evolve([D[II)V:8"/>

        </jPACE:loop>

        <jPACE:bytecodeBlock id="evolve([D[II)V:9"/>

      </jPACE:loop>

      <jPACE:bytecodeBlock id="evolve([D[II)V:10"/>

    </jPACE:loop>


  </jPACE:method>

  <jPACE:method class="java.lang.Math"
                method="exp" descriptor="(D)D" type="transaction">
  </jPACE:method>

  <jPACE:method class="uk.ac.warwick.dcs.hpsg.applications.dhpc.fft.DHPC_FFTBench"
                method="map" descriptor="(III)D" type="characterised">

    <jPACE:bytecodeBlock id="map(III)D:1"/>

    <jPACE:case>
      <jPACE:probValue value="0.5">

        <jPACE:bytecodeBlock id="map(III)D:2"/>

      </jPACE:probValue>
      <jPACE:probValue value="0.5">

        <jPACE:bytecodeBlock id="map(III)D:3"/>

      </jPACE:probValue>
    </jPACE:case>
```



```
    <jPACE:bytecodeBlock id="map(III)D:4"/>

    <jPACE:case>
      <jPACE:probValue value="0.5">

        <jPACE:bytecodeBlock id="map(III)D:5"/>

      </jPACE:probValue>
      <jPACE:probValue value="0.5">

        <jPACE:bytecodeBlock id="map(III)D:6"/>

      </jPACE:probValue>
    </jPACE:case>

    <jPACE:bytecodeBlock id="map(III)D:7"/>

    <jPACE:case>
      <jPACE:probValue value="0.5">

        <jPACE:bytecodeBlock id="map(III)D:8"/>

      </jPACE:probValue>
      <jPACE:probValue value="0.5">

        <jPACE:bytecodeBlock id="map(III)D:9"/>

      </jPACE:probValue>
    </jPACE:case>

    <jPACE:bytecodeBlock id="map(III)D:10"/>

  </jPACE:method>

  <jPACE:method class="uk.ac.warwick.dcs.hpsg.applications.dhpc.fft.DHPC_FFTBench"
                method="complex_fouriernd" descriptor="([D[I)V" type="characterised">

    <jPACE:bytecodeBlock id="complex_fouriernd([D[I)V:1"/>
    <jPACE:callMethod class="uk.ac.warwick.dcs.hpsg.applications.dhpc.fft.DHPC_FFTBench"
                      method="auxiliary_complex_fouriernd" descriptor="([D[II)V"/>

  </jPACE:method>

  <jPACE:method class="uk.ac.warwick.dcs.hpsg.applications.dhpc.fft.DHPC_FFTBench"
                method="auxiliary_complex_fouriernd" descriptor="([D[II)V" type="characterised">
```

```
    <jPACE:bytecodeBlock id="auxiliary_complex_fouriernd([D[II)V:1"/>

    <jPACE:loop count="3">

      <jPACE:bytecodeBlock id="auxiliary_complex_fouriernd([D[II)V:2"/>

    </jPACE:loop>

    <jPACE:bytecodeBlock id="auxiliary_complex_fouriernd([D[II)V:3"/>

    <jPACE:loop count="3">

      <jPACE:bytecodeBlock id="auxiliary_complex_fouriernd([D[II)V:4"/>

      <jPACE:loop count="(${n1} + ${n2} + ${n3}) / 3">

        <jPACE:bytecodeBlock id="auxiliary_complex_fouriernd([D[II)V:5"/>

        <jPACE:case>
          <jPACE:probValue value="0.4356" data_dependent="yes">

            <jPACE:bytecodeBlock id="auxiliary_complex_fouriernd([D[II)V:6"/>

            <jPACE:loop count="(1 + ${n2} + (${n2} * ${n1})) / 3">

              <jPACE:bytecodeBlock id="auxiliary_complex_fouriernd([D[II)V:7"/>

              <jPACE:loop count="3.4438" data_dependent="yes">

                <jPACE:bytecodeBlock id="auxiliary_complex_fouriernd([D[II)V:8"/
>

              </jPACE:loop>

              <jPACE:bytecodeBlock id="auxiliary_complex_fouriernd([D[II)V:9"/>

            </jPACE:loop>

          </jPACE:probValue>
        </jPACE:case>

        <jPACE:bytecodeBlock id="auxiliary_complex_fouriernd([D[II)V:10"/>

        <jPACE:loop count="0.5" data_dependent="yes">

          <jPACE:bytecodeBlock id="auxiliary_complex_fouriernd([D[II)V:11"/>

        </jPACE:loop>

        <jPACE:bytecodeBlock id="auxiliary_complex_fouriernd([D[II)V:12"/>
```



```
      </jPACE:loop>

      <jPACE:bytecodeBlock id="auxiliary_complex_fouriernd([D[II)V:13"/>

      <jPACE:loop count="((log(${n1}) / log(2)) + (log(${n2}) / log(2)) + (log(${
n3}) / log(2))) / 3">

        <jPACE:bytecodeBlock id="auxiliary_complex_fouriernd([D[II)V:14"/>
        <jPACE:callMethod class="java.lang.Math"
                          method="sin" descriptor="(D)D"/>
        <jPACE:bytecodeBlock id="auxiliary_complex_fouriernd([D[II)V:15"/>
        <jPACE:callMethod class="java.lang.Math"
                          method="sin" descriptor="(D)D"/>
        <jPACE:bytecodeBlock id="auxiliary_complex_fouriernd([D[II)V:16"/>

        <jPACE:loop count="(2^((((log(${n1}) / log(2)) + (log(${n2}) / log(2)) +
 (log(${n3}) / log(2))) / 3)) - 1) / (((log(${n1}) / log(2)) + (log(${n2}) / log
(2)) + (log(${n3}) / log(2))) / 3)">

          <jPACE:bytecodeBlock id="auxiliary_complex_fouriernd([D[II)V:17"/>

          <jPACE:loop count="(1 + ${n2} + (${n2} * ${n1})) / 3">

            <jPACE:bytecodeBlock id="auxiliary_complex_fouriernd([D[II)V:18"/>

            <jPACE:loop count="1.5 * (((log(${n1}) / log(2)) + (log(${n2}) / log
(2)) + (log(${n3}) / log(2))) / 3)">

              <jPACE:bytecodeBlock id="auxiliary_complex_fouriernd([D[II)V:19"/>

            </jPACE:loop>

            <jPACE:bytecodeBlock id="auxiliary_complex_fouriernd([D[II)V:20"/>

          </jPACE:loop>

          <jPACE:bytecodeBlock id="auxiliary_complex_fouriernd([D[II)V:21"/>

        </jPACE:loop>

        <jPACE:bytecodeBlock id="auxiliary_complex_fouriernd([D[II)V:22"/>

      </jPACE:loop>

      <jPACE:bytecodeBlock id="auxiliary_complex_fouriernd([D[II)V:23"/>

    </jPACE:loop>


  </jPACE:method>

  <jPACE:method class="java.lang.Math"
```

```
                method="sin" descriptor="(D)D" type="transaction">
  </jPACE:method>

  <jPACE:method class="uk.ac.warwick.dcs.hpsg.applications.dhpc.fft.DHPC_FFTBenc
h"
                method="next" descriptor="()D" type="characterised">

    <jPACE:bytecodeBlock id="next()D:1"/>
    <jPACE:callMethod class="uk.ac.warwick.dcs.hpsg.applications.dhpc.fft.DHPC_F
FTBench"
                      method="mult" descriptor="(DD)D"/>
    <jPACE:bytecodeBlock id="next()D:2"/>

  </jPACE:method>

  <jPACE:method class="uk.ac.warwick.dcs.hpsg.applications.dhpc.fft.DHPC_FFTBenc
h"
                method="mult" descriptor="(DD)D" type="characterised">

    <jPACE:bytecodeBlock id="mult(DD)D:1"/>

  </jPACE:method>

  <jPACE:method class="uk.ac.warwick.dcs.hpsg.applications.dhpc.fft.DHPC_FFTBenc
h"
                method="JGFkernel" descriptor="()V" type="characterised">

    <jPACE:bytecodeBlock id="JGFkernel()V:1"/>
    <jPACE:callMethod class="uk.ac.warwick.dcs.hpsg.applications.dhpc.fft.DHPC_F
FTBench"
                      method="doFFT" descriptor="()V"/>

  </jPACE:method>

</jPACE:transaction>
```

Appendix C

Platform Bytecode Block Timings

The following tables contain the bytecode block timings for every bytecode block that was defined during the automated characterisation of the five JavaGrande benchmarks. Each block contains an execution time for the 'mscs', 'labvista' and 'budweiser' workstations, as well as the difference in execution time before and after the virtual machine's adpative optimisation. All times are in nanoseconds.

'JGFSparseMatmultBench' class

Bytecode Block	'mscs'		'labvista'		'budweiser'	
	Before	After	Before	After	Before	After
JGFinitialise()V:1	572.004	39.303	1202.489	110.714	7662.477	617.38
JGFinitialise()V:10	339.3	36.914	629.404	48.141	3874.125	627.896
JGFinitialise()V:11	66.375	1	158.066	24.084	740.365	1
JGFinitialise()V:12	330.638	41.428	650.098	52.221	3768.038	1604.741
JGFinitialise()V:13	258.517	77.362	495.502	123.598	4006.579	446.637
JGFinitialise()V:14	152.747	4.464	359.893	121.196	2644.57	863.933
JGFinitialise()V:15	136.192	61.092	368.228	38.622	10866.445	211.005
JGFinitialise()V:16	77.44	4.113	158.037	24.653	974.464	1
JGFinitialise()V:17	125.924	49.303	217.7	36.389	1740.864	553.021
JGFinitialise()V:18	199.908	66.426	390.94	11.988	3189.875	157.97
JGFinitialise()V:19	66.972	15.89	182.756	26.476	815.59	168.684
JGFinitialise()V:2	373.049	9.277	755.001	128.265	6294.989	417.408
JGFinitialise()V:20	356.48	31.578	599.922	87.205	5592.704	453.606

Bytecode Block	'macs' Before	'macs' After	'labvista' Before	'labvista' After	'budweiser' Before	'budweiser' After
JGFinitialise()V:21	80.896	1	161.081	6.411	1045.85	101.402
JGFinitialise()V:22	350.763	31.108	727.737	64.859	3881.677	484.49
JGFinitialise()V:23	97.92	81	166.386	6.847	2797.043	1137.239
JGFinitialise()V:24	274.574	64.331	482.204	25.082	3277.184	489.567
JGFinitialise()V:25	274.603	12.656	490.197	32.674	3431.936	1089.715
JGFinitialise()V:26	274.574	102.677	727.609	29.193	3237.747	396.038
JGFinitialise()V:27	116.338	1	375.666	43.614	1613.888	872.471
JGFinitialise()V:28	137.956	14.255	201.842	131.766	1827.917	233.938
JGFinitialise()V:29	271.246	20.114	524.146	94.784	2897.664	336.21
JGFinitialise()V:3	196.395	63.62	439.012	131.507	5606.144	354.048
JGFinitialise()V:30	278.841	65.833	468.935	16.454	2812.864	251.008
JGFinitialise()V:31	280.548	57.481	574.066	10.121	3394.854	387.622
JGFinitialise()V:32	33.323	12.264	45.966	31.003	298.086	98.867
JGFinitialise()V:4	2755.385	226.194	7867.79	797.759	28564.966	8272.983
JGFinitialise()V:5	1952.441	236.153	5670.642	1304.37	40216.154	3736.036
JGFinitialise()V:6	77.468	2.724	164.594	136.809	2259.034	88.256
JGFinitialise()V:7	128.725	54.919	218.553	50.153	2309.978	295.191
JGFinitialise()V:8	147.925	12.591	225.792	25.748	1288.294	369.03
JGFinitialise()V:9	68.48	8.85	166.144	9.509	872.858	274.522
JGFkernel()V:1	176	21.441	336.341	7.108	5706.534	1689.28
RandomVector()[D:1	1105.166	175.041	3706.567	74.324	12426.163	2771.044
RandomVector()[D:2	139.42	16.843	199.538	1	1730.048	145.106
RandomVector()[D:3	98.162	61.012	279.893	11.689	1920.294	521.106
RandomVector()[D:4	30.848	65.714	168.818	29.51	333.21	488.941

'SparseMatmult' class

Bytecode Block	'macs' Before	'macs' After	'labvista' Before	'labvista' After	'budweiser' Before	'budweiser' After
test([D[D[I[I[D[I[D)V:1	243.866	88.9	410.652	7.514	2169.139	587.845
test([D[D[I[I[D[I[D)V:10	467.399	119.259	1052.999	138.395	5532.595	1209.085
test([D[D[I[I[D[I[D)V:2	177.408	38.289	266.112	23.542	1079.962	582.084
test([D[D[I[I[D[I[D)V:3	637.747	80.157	253.71	9.111	1273.613	190.199
test([D[D[I[I[D[I[D)V:4	582.554	283.635	1060.124	446.842	5734.976	1394.729
test([D[D[I[I[D[I[D)V:5	270.63	67.358	573.17	123.446	2944.883	1100.815
test([D[D[I[I[D[I[D)V:6	132.608	25.563	175.132	14.946	643.392	250.353
test([D[D[I[I[D[I[D)V:7	141.082	17.402	351.474	16.141	1615.514	806.577
test([D[D[I[I[D[I[D)V:8	140.378	70.562	179.911	42.315	1330.317	43.781
test([D[D[I[I[D[I[D)V:9	237.966	76.804	386.617	31.074	1517.235	426.883

'JGFSeriesBench' class

Bytecode Block	'macs'		'labvista'		'budweiser'	
	Before	After	Before	After	Before	After
Do()V:1	35.072	4.153	51.682	9.858	190.83	24.591
Do()V:2	275.541	2.873	504.934	9.238	2894.84	106.13
Do()V:3	1	1	1	1	1	1
Do()V:4	134.059	1	367.928	1	843.84	8.271
Do()V:5	35.925	8.789	98.35	9.783	329.035	28.651
Do()V:6	16.981	1	24.914	3.205	154.535	36.981
Do()V:7	95.744	2.759	197.785	9.91	942.398	94.494
Do()V:8	513.109	22.756	885.718	57.932	2901.682	232.849
Do()V:9	46.848	1	157.302	13.112	254.213	21.949
Do()V:10	94.549	4.466	249.082	14.032	782.348	94.398
Do()V:11	34.1	1	49.165	1	134.827	12.722
Do()V:12	23.125	1	82.52	5.4	325.095	94.01
Do()V:13	155.989	1	292.335	2.93	896.879	82.921
Do()V:14	362.581	7.253	462.185	23.421	1126.93	79.253
Do()V:15	123.307	3.214	432.095	7.15	1723.923	83.19
Do()V:16	31.829	9.273	130.1	16.111	606.221	41.9
Do()V:17	58.027	12.544	318.106	6.982	693.222	26.826
Do()V:18	117.931	1.394	238.054	8.532	895.2	39.43
Do()V:19	1	3.527	1	1	9.91	1
Do()V:20	112.384	2.162	143.973	9.042	943.91	73.162
Do()V:21	426.24	3.271	712.04	9.091	2034.91	92.980
Do()V:22	91.648	1	181.328	9.501	309.648	21.192
Do()V:23	72.875	2.361	105.293	4.073	271.091	23.421
Do()V:24	121.259	8.164	150.42	18.678	1422.605	35.955
Do()V:25	91.989	6.37	250.515	9.436	553.002	21.173
Do()V:26	37.547	1	39.454	1	89.511	7.534
JGFkernel()V:1	65.28	14.108	80.489	24.101	193.91	42.012

'SeriesTest' class

Bytecode Block	'macs'		'labvista'		'budweiser'	
	Before	After	Before	After	Before	After
thefunction(DDI)D:1	21.504	1	56.132	2.210	130.932	9.12
thefunction(DDI)D:2	75.605	1	179.421	4.012	501.430	45.320
thefunction(DDI)D:3	60.672	1	98.121	3.719	302.208	32.97
thefunction(DDI)D:4	52.395	3.035	123.973	15.035	555.320	65.079
thefunction(DDI)D:5	55.125	2.006	55.125	2.006	55.125	2.006

Bytecode Block	'msca'		'labvista'		' budweiser'	
	Before	After	Before	After	Before	After
thefunction(DDI)D:6	41.387	1.3	87.320	12.083	491.387	18.301
thefunction(DDI)D:7	71.595	1	130.812	3.938	690.321	43.921
thefunction(DDI)D:8	31.659	2.146	41.02	4.092	139.024	9.412
thefunction(DDI)D:9	32.427	1.339	32.938	2.013	290.321	13.723
thefunction(DDI)D:10	1	1	1	1	1	1
thefunction(DDI)D:11	1	1	1	1	1	1
TrapezoidIntegrate(DDIDI)D:1	253.952	7.623	223.089	8.547	1023.973	47.92
TrapezoidIntegrate(DDIDI)D:2	120.661	1	180.320	8.011	721.231	97.913
TrapezoidIntegrate(DDIDI)D:3	23.637	1	38.912	1	123.637	4.321
TrapezoidIntegrate(DDIDI)D:4	133.973	4.096	108.392	8.302	331.329	23.089
TrapezoidIntegrate(DDIDI)D:5	14.933	1	16.031	1	51.409	1.81
TrapezoidIntegrate(DDIDI)D:6	113.152	1.564	215.834	4.089	608.39	21.798
TrapezoidIntegrate(DDIDI)D:7	130.645	1	230.165	5.826	520.894	62.872

'JGFCryptBench' class

Bytecode Block	'msca'		'labvista'		' budweiser'	
	'Before	After	Before	After	Before	After
Do()V:1	125.284	6.841	240.796	16.411	1386.368	1
Do()V:10	93.596	1	196.338	28.496	1343.859	112.326
Do()V:11	39.58	3.433	60.103	146.441	104.371	857.085
Do()V:12	245.675	9.169	412.501	118.39	3965.107	805.769
Do()V:13	128.341	8.56	132.508	3.511	191.552	488.352
Do()V:14	138.51	50.16	233.088	37.508	1866.842	772.256
Do()V:15	137.984	7.529	334.649	276.712	2519.411	502.377
Do()V:16	145.052	63.659	241.678	2.741	1485.504	159.235
Do()V:17	102.884	2.076	148.85	26.933	2122.061	968.262
Do()V:18	209.508	68.864	240.185	11.88	1296.64	335.012
Do()V:19	186.14	56.235	351.474	117.736	3120.934	7449.533
Do()V:2	184.533	29.972	259.883	23.101	1982.528	319.089
Do()V:20	176.882	16.816	308.736	20.598	2929.613	1044.259
Do()V:21	265.472	58.695	468.366	32.337	3282.035	196.298
Do()V:22	126.436	46.292	316.274	15.923	1422.118	697.709
Do()V:23	73.515	19.632	66.048	141.126	178.97	148.644
Do()V:24	260.907	68.523	406.044	12.169	2948.608	691.855
Do()V:25	127.744	2.765	225.465	13.322	2237.158	133.245
Do()V:3	138.311	87.349	353.934	131.318	1422.605	593.027
Do()V:4	178.574	10.222	432.768	116.669	3028.659	272.707
Do()V:5	141.525	23.211	295.538	18.896	1828.16	642.669

Bytecode Block	'mscs'		'labvista'		' budweiser'	
	Before	After	Before	After	Before	After
Do()V:6	137.301	75.063	279.481	16.161	2051.341	244.448
Do()V:7	55.239	1	135.708	3.816	770.995	41.683
Do()V:8	45.824	1	85.305	29.747	606.221	1
Do()V:9	275.172	61.521	462.564	75.116	4596.749	689.334
JGFkernel()V:1	81.365	7.694	175.047	1	412.416	72.201

'IDEATest' class

Bytecode Block	'mscs'		'labvista'		' budweiser'	
	Before	After	Before	After	Before	After
cipher_idea([B[B[I)V:1	172.058	53.649	532.622	17.006	2360.538	693.887
cipher_idea([B[B[I)V:2	1766.118	561.129	4503.282	907.886	22387.93	4196.211
cipher_idea([B[B[I)V:3	1273.766	445.878	3302.585	942.346	15357.184	9715.886
cipher_idea([B[B[I)V:4	1148.314	320.147	2797.54	647.292	13795.661	3579.354

'DHPC_EPBench' class

Bytecode Block	'mscs'		'labvista'		' budweiser'	
	Before	After	Before	After	Before	After
ep()V:1	1197.881	13.837	3687.979	61.833	13001.434	330.776
ep()V:10	60.018	1	104.732	1	1114.33	1
ep()V:11	35.442	1	121.899	1	505.715	1
ep()V:12	25.444	1	114.261	1	532.518	1
ep()V:13	323.726	4.345	727.111	27.6	3333.043	1670.26
ep()V:14	63.815	1	213.291	1	932.544	1
ep()V:2	106.439	9.229	353.636	1	1328.371	96.396
ep()V:3	103.253	1	282.98	14.026	1565.274	31.743
ep()V:4	99.598	1	196.324	1	766.221	1
ep()V:5	268.174	3.696	354.887	1	1367.053	2.673
ep()V:6	563.584	2	660.196	1	2934.63	187.34
ep()V:7	73.543	1	158.734	1	956.416	1
ep()V:8	254.606	3.932	264.064	9.813	1576.358	1
ep()V:9	644.167	54.067	637.284	25.873	2944.55	76.504
JGFkernel()V:1	58.724	1	117.134	1	1	1

'KernelEP' class

Bytecode Block	'macs'		'labvista'		' budweiser'	
	Before	After	Before	After	Before	After
next()D:1	79.673	1	136.604	8.448	1260.198	1
next()D:2	387.243	1	651.819	27.708	2965.44	35.867
next()D:3	500.793	3.227	795.307	16.842	4702.067	98.423
next()D:4	92.132	1	229.276	11.059	839.642	66.049
next()D:5	486.983	1	785.479	8.451	4479.13	21.784
next()D:6	24.135	1	143.161	1	701.734	1
next()D:7	3041.252	181.925	3469.98	331.59	10703.974	2138.327

'DHPC_FFTBench' class

Bytecode Block	'macs'		'labvista'		' budweiser'	
	Before	After	Before	After	Before	After
auxiliary_complex_fouriernd([D[II)V:1	222.066	1	484.723	103.201	1980.582	128.502
auxiliary_complex_fouriernd([D[II)V:10	182.926	81.459	457.843	127.56	1853.414	56.617
auxiliary_complex_fouriernd([D[II)V:11	203.79	4.446	463.654	127.124	1922.957	5.66
auxiliary_complex_fouriernd([D[II)V:12	125.639	1	266.752	91.592	1902.771	98.166
auxiliary_complex_fouriernd([D[II)V:13	133.959	59.884	235.021	223.048	968	20.841
auxiliary_complex_fouriernd([D[II)V:14	682.382	27.984	957.914	238.561	5183.322	125.711
auxiliary_complex_fouriernd([D[II)V:15	388.708	9.068	389.056	137.723	2178.982	492.252
auxiliary_complex_fouriernd([D[II)V:16	175.744	1.729	333.146	122.427	1893.453	419.74
auxiliary_complex_fouriernd([D[II)V:17	167.893	2.17	371.085	71.624	1802.342	256.271
auxiliary_complex_fouriernd([D[II)V:18	119.751	1	221.862	100.014	1248.282	233.59
auxiliary_complex_fouriernd([D[II)V:19	1148.288	27.116	1770.266	278.856	8060.787	557.967
auxiliary_complex_fouriernd([D[II)V:2	208.782	8.641	461.645	246.587	2265.498	220.009
auxiliary_complex_fouriernd([D[II)V:20	128.796	9.082	289.856	101.14	1324.39	249.219
auxiliary_complex_fouriernd([D[II)V:21	921.415	17.63	863.578	188.014	3435.008	523.779
auxiliary_complex_fouriernd([D[II)V:22	95.417	1	237.005	93.46	912.166	244.521
auxiliary_complex_fouriernd([D[II)V:23	159.758	5.427	288.563	94.049	1746.47	265.411
auxiliary_complex_fouriernd([D[II)V:3	156.686	1.26	338.816	81.825	1656.525	289.91
auxiliary_complex_fouriernd([D[II)V:4	431.644	9.722	1063.693	228.001	4418.509	61.571
auxiliary_complex_fouriernd([D[II)V:5	74.041	1	152.333	59.22	599.821	1171.331
auxiliary_complex_fouriernd([D[II)V:6	159.886	6.238	356.032	96.008	1703.514	772.367
auxiliary_complex_fouriernd([D[II)V:7	124.174	6.209	198.157	81.236	1204.045	569.039
auxiliary_complex_fouriernd([D[II)V:8	501.66	1.872	1097.357	249.185	5737.638	864.067
auxiliary_complex_fouriernd([D[II)V:9	108.046	63.582	311.808	74.568	1160.998	264.374

Bytecode Block	'mace'		'labvista'		'budweiser'	
	Before	After	Before	After	Before	After
complex_fouriernd([D[I)V:1	140.43	10.433	214.976	66.67	1335.936	1
doFFT()V:1	2283.506	210.341	6552.55	1288.251	22234.867	2548.764
doFFT()V:10	45.881	3.521	126.451	65.518	275.29	93.084
doFFT()V:2	144.725	1	370.458	88.955	1755.968	1
doFFT()V:3	45.724	68.659	199.846	85.23	1362.88	72.809
doFFT()V:4	100.964	1	232.166	59.553	1742.003	446.492
doFFT()V:5	106.482	17.729	193.101	78.932	952.858	191.439
doFFT()V:6	156.9	25.68	362.163	95.112	2240.371	1
doFFT()V:7	115.541	1	211.059	71.06	1920.794	1
doFFT()V:8	175.502	8.826	344.55	140.756	1573.773	143.759
doFFT()V:9	934.5	71.077	1633.178	383.47	8259.123	767.209
evolve([D[II)V:1	673.593	36.432	828.749	245.883	4964.058	166.825
evolve([D[II)V:10	111.417	1	236.006	89.966	1479.066	1
evolve([D[II)V:2	141.924	11.913	298.048	106.12	2030.067	68.572
evolve([D[II)V:3	150.485	1.26	293.581	102.766	2126.656	665.948
evolve([D[II)V:4	370.759	5.683	659.085	129.364	5634.266	193.59
evolve([D[II)V:5	180.636	1	180.211	77.435	1223.283	1
evolve([D[II)V:6	291.84	5.626	644.019	151.188	4280.538	169.487
evolve([D[II)V:7	155.79	1	196.173	98.862	1241.83	1
evolve([D[II)V:8	269.554	1	594.342	142.459	3643.098	235.932
evolve([D[II)V:9	116.423	1	249.562	95.419	1432.154	1
inverse_complex_fouriernd([D[I)V:1	119.993	10.391	213.03	86.318	923.341	1
inverse_complex_fouriernd([D[I)V:2	295.993	1.417	619.981	118.779	2474.522	727.491
inverse_complex_fouriernd([D[I)V:3	191.019	3.322	592.013	151.88	3422.682	416.425
JGFkernel()V:1	97.28	1	151.974	55.355	422.157	1
map(III)D:1	137.572	17.886	318.106	92.539	2283.238	38.684
map(III)D:10	240.882	10.817	351.654	97.787	1931.085	1468.342
map(III)D:2	30.45	2.497	65.818	70.344	217.907	1
map(III)D:3	104.932	1	149.658	78.036	1540.237	96.105
map(III)D:4	173.653	4.915	370.419	220.91	2459.93	237.519
map(III)D:5	46.578	13.577	49.165	170.324	220.173	1
map(III)D:6	95.26	1	157.184	67.771	1689.971	1
map(III)D:7	1369.529	968.405	374.976	136.929	5378.586	135.273
map(III)D:8	1265.863	944.242	38.784	62.024	322.662	1
map(III)D:9	1322.738	944.213	167.014	87.432	1516.403	193.436
mult(DD)D:1	3987.911	985.6	3128.333	495.15	8951.309	3252.419
next()D:1	1436.871	928.455	432	84.424	2563.072	227.331
next()D:2	1457.55	953.643	486.451	108.782	3108.621	101.161

References

[Addison93] C. Addison, V. Getov, A. Hey, R. Hockney and I. Wolton. 'The GENESIS Distributed-Memory Benchmarks.' In J. Dongarra and W. Gentzsch, editors, Proceedings of the Computer Benchmarks, pp. 257–271. North Holland (1993).

[Aida00] K. Aida, A. Takefusa, H. Nakada, S. Matsuoka, S. Sekiguchi and U. Nagashima. 'Performance Evaluation Model for Scheduling in Global Computing System.' The International Journal of High Performance Computing Applications, **14**(3):pp. 268–279 (Fall 2000).

[Alkindi00] A. Alkindi, D. Kerbyson, E. Papaefstathiou and G. Nudd. 'Runtime Optimisation Using Dynamic Performance Prediction.' High Performance Computing and Networking, Lecture Notes in Computer Science, **1923**:pp. 280–289 (May 2000).

[Apache03] Apache. BCEL: The Bytecode Engineering Library. The Apache Software Foundation (2003).
URL `http://jakarta.apache.org/bcel/index.html`

[Arnold01] M. Arnold and B. Ryder. 'A Framework for Reducing the Cost of Instrumented Code.' In Proceedings of the SIGPLAN Conference on Programming Language Design and Implementation, pp. 168–179. Utah, USA (June 2001).

[Bacigalupo03a] D. Bacigalupo, J. Turner, S. Jarvis, D. Dillenberger and G. Nudd. 'A Dynamic Predictive Framework for e-business Workload Management.' In Proceedings of the 7th World Multiconference on Systemics, Cybernetics and Informatics. Orlando, USA (July 2003).

[Bacigalupo03b] D. Bacigalupo, J. Turner, S. Jarvis and G. Nudd. 'Dynamic Workload Management using SLAs and an e-Business Performance Prediction Framework.' (October 2003).

[Bagrodia98] R. Bagrodia, R. Meyer, M. Takai, Y. Chen, X. Zeng, J. Martin and H. Song. 'Parsec: A Parallel Simulation Environment for Complex Systems.' IEEE Computer, **31**(10):pp. 75–85 (1998).

[Bailey91] D. H. Bailey, E. Barszcz, J. T. Barton, D. S. Browning, R. L. Carter, D. Dagum, R. A. Fatoohi, P. O. Frederickson, T. A. Lasinski, R. S. Schreiber, H. D. Simon, V. Venkatakrishnan and S. K. Weeratunga. 'The NAS Parallel Benchmarks.' The International Journal of Supercomputer Applications, **5**(3):pp. 63–73 (Fall 1991).

[Ben-Naten95] R. Ben-Naten. CORBA: A Guide to Common Object Request Broker Architecture. McGraw-Hill, New York, USA (1995).

[Berkbigler03] K. Berkbigler, B. Bush, K. Davis, N. Moss, S. Smith, T. Caudell, K. Summers and C. Zhou. 'A la carte: A Simulation Framework for Extreme-Scale Hardware Architectures.' In Proceedings of the International Conference on Modelling and Simulation. Palm Springs, USA (February 2003).

[Bull00] J. Bull, L. Smith, M. Westhead, D. Henty and R. Davey. 'A Benchmark Suite for High Performance Java.' Concurrency: Practice and Experience, **12**:pp. 357–388 (2000).

[Buyya00] R. Buyya, D. Abramson and J. Giddy. 'Nimrod/G: An Architecture for a Resource Management and Scheduling System in a Global Computational Grid.' In Proceedings of the 4th International Conference and Exhibition on High Performance Computing in Asia-Pacific Region. IEEE Computer Society Press, 2000, Beijing, China (May 2000).

[Buyya02] R. Buyya and M. Murshed. 'GridSim: A Toolkit for the Modeling and Simulation of Distributed Resource Management and Scheduling for Grid Computing.' Journal of Concurrency and Computation: Practice and Experience, **14**:pp. 1–32 (December 2002).

[Cao99] J. Cao, D. Kerbyson, E.Papaefstathiou and G. Nudd. 'Modelling of ASCI High Performance Applications using PACE.' In Proceedings of the UK Performance Engineering Workshop, pp. 413–424. Bristol, UK (July 1999).

[Cao00] J. Cao, D. Kerbyson, E. Papaefstathiou and G. Nudd. 'Modelling of ASCI High Performance Applications using PACE.' In Proceedings of the 19th IEEE International Performance, Computing and Communication Conference, pp. 485–492. Phoenix, USA (2000).

[Cao01a] J. Cao. Agent-based Resource Management for Grid Computing. Ph.D. thesis, Dept. Computer Science, University of Warwick, UK (October 2001).

[Cao01b] J. Cao, D. Kerbyson and G. Nudd. 'High Performance Service Discovery in Large-scale Multi-agent and Mobile-agent Systems.' International Journal of Software Engineering and Knowledge Engineering, Special Issue on Multi-Agent Systems and Mobile Agents, **11**(5):pp. 621–641 (2001).

[Cao01c] J. Cao, D. Kerbyson and G. Nudd. 'Performance Evaluation of an Agent-Based Resource Management Infrastructure for Grid Computing.' In Proceedings of the 1st IEEE/ACM International Symposium on Cluster Computing and the Grid, pp. 311–318. Brisbane, Australia (May 2001).

[Cao02a] J. Cao, S. Jarvis, D. Spooner, J. Turner and G. Nudd. 'Performance Prediction Technology for Agent-based Resource Management.' In Proceedings of the 11th IEEE Heterogeneous Computing Workshop. Fort Lauderdale, USA (April 2002).

[Cao02b] J. Cao, D. Spooner, J. Turner, S. Jarvis, D. Kerbyson, S. Saini and G. Nudd. 'Agent-based Resource Management for Grid Computing.' In Proceedings of the 2nd IEEE/ACM International Symposium on Cluster Computing and the Grid. Berlin, Germany (May 2002).

[Cao03] J. Cao, S. Jarvis, S. Saini and G. Nudd. 'GridFlow: Workflow Management for Grid Computing.' In Proceedings of the 3rd IEEE/ACM International Symposium on Cluster Computing and the Grid. Tokyo, Japan (May 2003).

[Casanova01] H. Casanova. 'Simgrid: A Toolkit for the Simulation of Application Scheduling.' In Proceedings of the 1st IEEE/ACM

Interational Symposium on Cluster Computing and the Grid. Brisbane, Australia (May 2001).

[Covington88] R. Covington, S. Madala, V. Metha, J. Jump and J. Sinclair. 'The Rice Parallel Processing Testbed.' ACM Sigmetrics Conference on Measurement and Modelling of Computer Systems, pp. 4–11 (1988).

[Covington91] R. Covington, S. Dwarkadas, J. Jump, J. Sinclair and S. Madala. 'Efficient Simulation of Parallel Computer Systems.' International Journal in Computer Simulation, 1(1):pp. 32–58 (1991).

[Cowie99] J. Cowie, H. Liu, J. Liu, D. Nicol and A. Ogielski. 'Towards Realistic Million-Node Internet Simulations.' In Proceedings of the 1999 International Conference on Parallel and Distributed Processing Techniques and Applications. Las Vegas, USA (July 1999).

[Culler93] D. Culler, R. Karp, D. Patterson, A. Sahay, K. Schauser, E. Santos, R. Subramonian and T. VonEiken. 'LogP: Towards a Realistic Model of Parallel Computation.' In Proceedings of the 4th ACM SIGPLAN Symposium on Principles and Practices of Parallel Programming, pp. 1–12. San Diego, USA (May 1993).

[Deelman98] E. Deelman, A. Dube, A. Hoisie, Y. Luo, R. Oliver, D. Sundaram-Stukel, H. Wasserman, V. Adve, R. Bagrodia, J. Browne, E. Houstis, O. Lubeck, J. Rice, P. Teller and M. Vernon. 'POEMS: End-to-end Performance Design of Large Parallel Adaptive Computational Systems.' In Proceedings of the 1st

ACM International Workshop on Software and Performance, pp. 18–30. Sante Fe, USA (October 1998).

[DeRose98] L. DeRose, Y. Zhang and D. Reed. 'SvPablo: A Multi-Language Performance Analysis System.' In Proceedings of the 10th International Conference on Computer Performance, pp. 252–255. Spain (1998).

[Dixit93] K. Dixit. 'The SPEC benchmarks.' In J. Dongarra and W. Gentzsch, editors, Computer Benchmarks, pp. 149–163. North Holland (1993).

[Dongarra93] J. Dongarra and H. Vorst. 'Performance of Various Computers using Standard Sparse Linear Equations Solving Techniques.' In J. Dongarra and W. Gentzsch, editors, Computer Benchmarks, pp. 177–188. North Holland (1993).

[Foster97] I. Foster and C. Kesselman. 'A Metacomputing Infrastructure Toolkit.' International Journal of Supercomputer Applications, **11**(2):pp. 115–128 (1997).

[Foster98] I. Foster and C. Kesselman. The GRID: Blueprint for a New Computing Infrastructure. Morgan-Kaufman (1998).

[Foster01] I. Foster, C. Kesselman and S. Tuecke. 'The Anatomoy of the Grid: Enabling Scalable Virtual Organisations.' International Journal of Supercomputing Applications, **15**(3) (2001).

[Foster02a] I. Foster, C. Kesselman, J. Nick and S. Tuecke. 'The Physiology of the Grid: An Open Grid Services Architecture for Distributed Systems Integration.' In Proceedings of the Open Grid Service Infrastructure WG, Global Grid Forum (June 2002).

[Foster02b] I. Foster, A. Roy and V. Sander. 'Grid Services for Distributed System Integration.' IEEE Computer, **35**(6):pp. 37–46 (2002).

[Frank97] M. Frank, A. Agarwal and M. Vernon. 'LoPC: Modeling Contention in Parallel Algorithms.' In Proceedings of the 6th ACM SIGPLAN Symposium on Principles and Practices of Parallel Programming, pp. 276–287. Las Vegas, USA (June 1997).

[Funk03] N. Funk. JEP: Java Math Expression Parser Documentation (2003).
URL `http://www.singularsys.com/jep/doc/index.html`

[Getov01] V. Getov, G. von Laszewski, M. Philippsen and I. Foster. 'Multiparadigm Communications in Java for Grid Computing.' Communications of the ACM, **44**(10):pp. 118–125 (October 2001).

[Gilbert98] J. Gilbert and R. Brodersen. 'A Lossless 2-D Image Compression Technique for Synthetic Discrete-Tone Images.' IEE DCC, pp. 359–368 (1998).

[Gosling00] J. Gosling, B. Joy, G. Steele and G. Bracha. The Java Language Specification. Sun Microsystems, 2nd edition (2000).
URL `http://java.sun.com/docs/books/jls`

[Gropp96a] W. Gropp, E. Lusk, N. Doss and A. Skjellum. 'A High-performance, Portable Implementation of the MPI Message Passing Interface Standard.' Parallel Computing, **22**(6):pp. 789–828 (1996).

[Gropp96b] W. D. Gropp and E. Lusk. User's Guide for mpich, a Portable Implementation of MPI. Mathematics and Computer Science Division, Argonne National Laboratory (1996). ANL-96/6.

[Gunter00a] D. Gunter and W. Smith. 'Schemas for Grid Performance Events.' Technical report, The Global Grid Forum (2000). URL http://www-didc.lbl.gov/GGF-PERF/GMA-WG

[Gunter00b] D. Gunter, D. Tierney, B. Crowley, M. Holding and J. Lee. 'NetLogger: A Toolkit for Distributed System Performance Analysis.' In Proceedings of the IEE Masctos 2000 Conference (2000).

[Gupta00] R. K. Gupta. REVEAL: A Distributed Class Decompiler to Facilitate Reverse Engineering in Java (2000). URL http://www.jreveal.org/paper.html

[Harper99] J. Harper, D. Kerbyson and G. Nudd. 'Analytical Modelling of Set-Associative Cache Behaviour.' IEEE Transactions on Computers, **49**(10):pp. 1009–1024 (October 1999).

[Herring90] C. Herring. 'A New Object-Oriented Simulation Language.' In Proceedings of the Object Oriented Simulation, pp. 55–60 (1990).

[Hockney94] R. Hockney and M. Berry. 'PARKBENCH report: Public International Benchmarks for Parallel Computers.' Scientific Programming, **3**(2):pp. 101–146 (1994).

[Hodges02] A. Hodges, G. Froeblich, D. Peercy, M. Pilch, J. Meza, M.-M. Peterson, J. LaGrange, L. Cox, K. Koch, N. Storch, C. Nitta and E. Dube. ASCI Software Quality Engineering: Goals, Princi-

ples and Guidelines (2002).
URL http://www.dp.doe.gov/asc/home.htm

[Howell98] F. Howell and R. McNab. 'Simjava: A Discrete Event Simulation Package for Java with Applications in Computer Systems Modelling.' In Proceedings of the 1st International Conference on Web-based Modelling and Simulation. San Diego, USA (Janurary 1998).

[HPSG99] HPSG. PACE Reference Manual. University of Warwick (1999).
URL http://www.dcs.warwick.ac.uk/ hpsg

[Huang01] E. Huang and T. Elrad. 'Reflective Decision Controls for Autonomous Distributed Objects.' In Proceedings of the 5th International Symposium on Autonomous Decentralized Systems, p. 212. Dallas, USA (March 2001).

[IBM03a] IBM. CFParse: A low-level API for interactive classFile editing (2003).
URL http://alphaworks.ibm.com/tech/cfparse

[IBM03b] IBM. 'IBM Web Services Toolkit.' (2003).
URL http://www.alphaworks.ibm.com/tech/webservicestoolkit

[Jain91] R. Jain. The Art of Computer Performance Analysis. John Wiley and Sons (1991).

[Jarvis03a] S. Jarvis, D. Spooner, H. L. C. Keung, J. Cao, S. Saini and G. Nudd. 'Performance Prediction and its use in Parallel and Distributed Computing Systems.' In Proceedings of

the IEEE/ACM International Workshop on Performance Modelling, Evaluation and Optimisation of Parallel and Distributed Systems. Nice, France (April 2003).

[Jarvis03b] S. Jarvis, D. Spooner, H. L. C. Keung, J. Dyson, L. Zhao and G. Nudd. 'Performance-based Middleware Services for Grid Computing.' In Proceedings of the 5th International Workshop on Active Middleware Services. Seattle, USA (June 2003).

[Johnson00] M. Johnson and J. Crowe. 'Measure the Performance of ARM 3.0 for Java.' In Proceedings of the Inte..ional Conference of the Computer Measurement Group. Orlando, USA (December 2000).

[Kerbyson96] D. Kerbyson, J. Harper, A. Craig and G. Nudd. 'PACE: A Toolset to Investigate and Predict Performance in Parallel Systems.' In Proceedings of the European Parallel Tools Meeting. Paris, France (October 1996).

[Kerbyson98a] D. Kerbyson and G. Nudd. 'Performance Prediction and Analysis using the PACE Toolset.' In Proceedings of the 13th RAPS Workshop. Daresbury Labs, UK (May 1998).

[Kerbyson98b] D. Kerbyson, E. Papaefstathiou and G. Nudd. 'Application Execution Steering using On-the-Fly Performance Prediction.' In Proceedings of High Performance Computing and Networking, volume 1401, pp. 718–727. Amsterdam, Holland (April 1998).

[Keung02a] H. L. C. Keung, J. Cao, D. Spooner, S. Jarvis and G. Nudd. 'Grid Information Services using Software Agents.' In Proceedings of the 18th Annual UK Performance Engineering

Workshop, pp. 187–198. University of Glasgow, UK (July 2002).

[Keung02b] H. L. C. Keung, L. Wang, D. Spooner, S. Jarvis, W. Jie and G. Nudd. 'Grid Resource Management Information Services for Scientific Computing.' In Proceedings of the International Conference on Scientific and Engineering Computation. Raffles City Covention Centre, Singapore (December 2002).

[Kielmann01] T. Kielmann, P. Hatcher, L. Bouge and H. Bal. 'Enabling Java for High-Performance Computing: Exploiting Distributed Shared Memory and Remote Method Invocation.' Communications of the ACM, **44**(10):pp. 110–117 (October 2001).

[Kon00] F. Kon, M. Roman, P. Liu, J. Mao, T. Yamane, L. Magalhaes and R. Campbell. 'Monitoring, Security, and Dynamic Configuration with the dynamicTAO Reflective ORB.' In Proceedings of the Middleware 2000 Conference. New York, USA (April 2000).

[Leg99] The Legion Research Group, University of Virginia, Charlottesville, VA. Legion 1.6 Developer Manual (1999).
URL `http://legion.virginia.edu`

[Leinberger99] W. Leinberger and V. Kumar. 'Information Power Grid: The New Frontier in Parallel Computing?' IEEE Concurrency, **7**(4) (1999).

[Lindholm99] T. Lindholm and F. Yellin. The Java Virtual Machine Specification. Sun Microsystems, 2nd edition (1999).
URL `http://java.sun.com/docs/books/vmspec`

[Litzkow88] M. Litzkow, M. Livny and M. Mutka. 'Condor - a Hunter of Idle Workstations.' In Proceedings of the 8th International Conference on Distributed Computing Systems, pp. 104–111 (1988).

[Liu02] J. Liu and D. Nicol. Dartmouth Scalable Simulation Framework User's Manual. Dartmouth College Dept. of Computer Science (February 2002).

[Lopez-Hernandez03] R. Lopez-Hernandez. SMC - A Lossless Compression Algorithm. Ph.D. thesis, Dept. Computer Science, University of Warwick, UK (Janurary 2003).

[Mathew99] J. Mathew, P. Coddington and K. Hawick. 'Analysis and Development of Java Grande Benchmarks.' Technical Report DHPC-063, Distributed High-Performance Computing Group, University of Adelaide, Austraila (1999).

[Meloan99] S. Meloan. 'The Java Hotspot Performance Engine: An In-Depth Look.' A `java.sun.com` Article (June 1999). URL `http://developer.java.sun.com/developer/technicalArticles/Networking/HotSpot/`

[Meyer97] J. Meyer and T. Downing. Java Virtual Machine. O'Reilly (March 1997).

[Miller95] B. Miller, M. Callaghan, J. Cargille, J. Hollingsworth, R. Irvin, K. Karavanic, K. Kunchithapadam and T. Newhall. 'The Paradyn Parallel Performance Measurement Tools.' IEEE Computer, **28**(11):pp. 37–46 (1995).

[Nicol02] D. Nicol and J. Liu. 'Composite Synchronisation in Parallel Discrete-Event Simulation.' IEEE Transactions on Parallel and Distributed Systems, **13**(5):pp. 433–446 (May 2002).

[Nudd93] G. Nudd, E. Papaefstathiou, Y. Papay, T. Atherton, C. Clarke, D. Kerbyson, A. Stratton, R. Ziani and M. Zemerly. 'A Layered Approach to the Characterisation of Parallel Systems for Performance Prediction.' In Proceedings of the Performance Evaluation of Parallel Systems, pp. 26–34. University of Warwick, UK (November 1993).

[Nudd00] G. Nudd, D. Kerbyson, E. Papaefstathiou, S. Perry, J. Harper and D. Wilcox. 'PACE: A Toolset for the Performance Prediction of Parallel and Distributed Systems.' International Journal of High Performance Computing Applications, Special Issue on Performance Modelling, **14**(3):pp. 228–251 (2000).

[OpenGroup01] OpenGroup. Application Response Measurement Issue 3.0 - Java Binding (2001).
URL `http://www.opengroup.org/management/arm.htm`

[Papaefstathiou88] E. Papaefstathiou. 'A Simulation Environment for Distributed/Parallel Systems.' Technical report, Digital Systems and Computers Lab, Physics Department, Thessaloniki, Greece (1988).

[Papaefstathiou94] E. Papaefstathiou, D. Kerbyson and G. Nudd. 'A Layered Approach to Parallel Software Prediction: A Case Study.' Massively Parallel Processing Applications and Development, pp. 617–624 (1994).

[Papaefstathiou95a] E. Papaefstathiou. A Framework for Characterising Parallel Systems for Performance Evaluation. Ph.D. thesis, Dept. Computer Science, University of Warwick, UK (1995).

[Papaefstathiou95b] E. Papaefstathiou, D. Kerbyson, G. Nudd and T. Atherton. 'An Overview of the CHIPS Performance Prediction Toolset for Parallel Systems.' In Proceedings of the 8th ISCA International Conference on Parallel and Distributed Computing Systems, pp. 527–533. Orlando, USA (September 1995).

[Papaefstathiou97] E. Papaefstathiou, D. Kerbyson, G. Nudd, T. Atherton and J. Harper. 'An Introduction to the Layered Characterisation for High Performance Systems.' Technical Report 335, Dept. Computer Science, University of Warwick, UK (1997).

[Pombortsis94] A. Pombortsis, E. Papaefstathiou, A. Beglis and G. Nudd. 'PASE: A Performance Analysis and Simulation Environment.' Simulation Practice and Theory, **2**:pp. 43–59 (1994).

[Prakash98] S. Prakash and R. Bagrodia. 'Using Parallel Simulation to Evalute MPI Programs.' In Proceedings of the Winter Simulation Conference. Washington DC, USA (December 1998).

[Project00] U. Project. 'UDDI Technical White Paper.' Technical report (2000).
URL `http://www.uddi.org`

[Ribler98] R. Ribler, J. Vetter, H. Simitci and D. Reed. 'Autopilot: Adaptive Control of Distributed Applications.' In Proceedings of the 7th IEEE Symposium on High-Performance Distributed Computing. Chicago, USA (July 1998).

[Rice79] R. Rice. 'Some Practical Universal Noiceless Coding Techniques.' Technical Report JPL-79-22, Jet Propulsion Laboratory, Pasadena, USA (1979).

[Rusell83] E. Rusell. 'Building Simulation Models: SimScript II.5.' Technical report, CACI Inc. (1983).

[Schneier96] B. Schneier. Applied Cryptography. John Wiley and Sons (1996).

[Smith90] C. Smith. Performance Engineering of Software Systems. Addison-Wesley (1990).

[Song00] H. Song, X. Liu, D. Jakobsen, R. Bhagwan, X. Zhang, K. Taura and A. Chien. 'The MicroGrid: a Scientific Tool for Modelling Computational Grids.' Scientific Programming, **8**(3):pp. 127–141 (2000).

[Spooner01] D. Spooner, J. Turner, J. Cao, S. Jarvis and G. Nudd. 'Application Characterisation using a Lightweight Transaction Model.' In Proceedings of the 17th Annual UK Performance Engineering Workshop, pp. 215–225. Leeds, UK (July 2001).

[Spooner02a] D. Spooner, J. Cao, J. Turner, H. L. C. Keung, S. Jarvis and G. Nudd. 'Localised Workload Management Using Performance Prediction and QoS Contracts.' In Proceedings of the 18th Annual UK Performance Engineering Workshop. University of Glasgow, UK (July 2002).

[Spooner02b] D. Spooner, S. Jarvis, J. Cao, G. Nudd, S. Saini and D. Kerbyson. 'Performance-based Workload Management for Grid Computing.' In Proceedings of the Los Alamos Computer Science Institute Symposium. Santa Fe, USA (October 2002).

[Spooner03] D. Spooner, S. Jarvis, J. Cao, S.Saini and G. Nudd. 'Local Grid Scheduling Techniques using Performance Prediction.' IEE Proceedings - Computers and Digital Techniques, **150**(2):pp. 87–96 (2003).

[Suganuma00] T. Suganuma, T. Ogasawara, M. Takeuchi, T. Yasue, M. Kawahito, K. Ishizaki, H. Komatsu and T. Nakatani. 'Overview of the IBM Java Just-In-Time Compiler.' IBM Systems Journal: Java Performance, **39**(1):pp. 175–193 (2000).

[Sun00] Sun. Codine 5.2 Manual, Revision A. Sun Microsystems, Palo Alto, USA (September 2000).
URL `http://www.sun.com/gridware`

[Sun02a] Sun. The Java HotSpot Virtual Machine, v1.4.1, d2. Sun Microsystems (September 2002).
URL `http://java.sun.com/products/hotspot`

[Sun02b] X. Sun, T. Fahringer and M. Pantano. 'SCALA: A Performance System for Scalable Computing.' International Journal of High Performance Computing Applications, **16**(4) (Autumn 2002).

[Tierney00] B. Tierney, B. Crowley, D. Gunter, M. Holding, J. Lee and M. Thompson. 'A Monitoring Sensor Management System for Grid Environments.' In Proceedings of the IEEE High Performance Distributed Computing Conference. Pittsburgh, USA (August 2000).

[Tierney01] B. Tierney, R. Aydt, D. Gunter, W. Smith, V. Taylor, R. Wolski and M. Swany. 'A Grid Monitoring Service Architecture.' White Paper, Grid Performance Working Group, Global Grid Forum (2001).

[Turner01] J. Turner, D. Spooner, J. Cao, S. Jarvis, D. Dillenberger and G. Nudd. 'A Transaction Definition Language for Application Response Measurement.' International Journal of Computer Resource Measurement, **105**:pp. 55–65 (Winter 2001).

[Turner02a] J. Turner, D. Bacigalupo, S. Jarvis, D. Dillenberger and G.R.Nudd. 'Application Response Measurement of Distributed Web Services.' International Journal of Computer Resource Management, **108**:pp. 45–55 (Autumn 2002).

[Turner02b] J. Turner, D. Bacigalupo, S. Jarvis and G. Nudd. 'Using a Transaction Definition Language for the Automated ARMing of Web Services.' In Proceedings of the UKCMG Annual Conference and Exibition. Reading, UK (May 2002).

[Turner02c] J. Turner, R. Lopez-Hernandez, D. Kerbyson and G. Nudd. 'Performance Optimisation of a Lossless Compression Algorithm using the PACE Toolkit.' Technical Report 398, Dept. Computer Science, University of Warwick, UK (2002).

[Uysal98] M. Uysal, T. Kurc, A. Sussman and J. Saltz. 'A Performance Prediction Framework for Data Intensive Applications on Large-Scale Parallel Machines.' In Proceedings of the 4th Workshop on Languages, Compilers and Run-time Systems for Scalable Computers, pp. 243–258. Pittsburgh, USA (May 1998).

[vanderSteen93] A. van der Steen. 'The Benchmark of the EuroBen Group.' Computer Benchmarks, pp. 165–175 (1993).

[vanVliet03] H. van Vliet. Mocha: the Java Decompiler (2003).
URL http://www.brouhaha.com/~eric/computers/mocha.html

[W3C00] W3C. Extensible Markup Language (XML) 1.0, 2nd edition (October 2000).
URL http://www.w3.org/TR/REC-xml

[Waheed00] A. Waheed, W. Smith, J. George and J. Yan. 'An Infrastructure for Monitoring and Management in Computational Grids.' In Proceedings of the 2000 Conference on Languages, Compilers and Runtime Systems. New York, USA (May 2000).

[Weinberger98] M. Weinberger, G. Seroussi and G. Sapiro. 'The LOCO-I Lossless Image Compression Algorithm: Principles and Standardization into JPEG-LS.' Technical report, Hewlett Packard Labs HPL (1998).

[Wijngaart02] R. V. D. Wijngaart and M. Frumkin. 'NAS Grid Benchmarks Version 1.0.' Technical Report NAS-02-005, NASA Ames Research Center (2002).

[Wolski99] R. Wolski, N. Spring and J. Hayes. 'The Network Weather Service: A Distributed Resource Performance Forecasting Service for Metacomputing.' Journal of Future Generation Computing Systems, **15**(5–6):pp. 757–768 (October 1999).

[Wu97] X. Wu and N. Memon. 'Context-based Adaptive Lossless Image Coding.' IEEE Transactions, **45**(4):pp. 437–444 (1997).